UNIVERSITY OF AMAR TELIDJI LAGHOUAT

FACULTY OF SCIENCES
COMPUTER SCIENCE DEPARTMENT

OPTION : INFORMATIQUE RÉPARTIE ET MOBILE

Thesis submitted in partial fulfillment of the requirements for the degree of doctor

TREE AUTOMATA ALGORITHMS AND APPLICATIONS

Written by	BELABBACI Ahlem	
Promoted by	Prof. CHERROUN Hadda	University of Laghouat
	Prof. ZIADI Djelloul	University of Rouen, France
Chairman	Prof. YAGOUBI Mohamed Bachir	University of Laghouat
Examined by	Prof. KAMEL Nadjet	University of Setif
	Prof. BENSAOU Nacira	University of Algiers

2018/2019

Acknowledgment

First, I would like to express my gratitude to Prof. Hadda Cherroun, Prof. Djelloul Ziadi for directing, correcting, and advising me throughout this work.

My thanks are extended to my defense committee for offering their time and good will throughout the review of this document: Prof. Yagoubi Mohammed Bachir, the chairman, and the examiners: Prof. Kamel Nadjat and Prof. Bensaou Nacira.

I would also like to extend my thanks to the members of the LIM laboratory who have supported, advised, and helped me in many ways during the various stages of this work especially Dr. Younes Guellouma.

Last but not least, a huge thank to all my family especially my parents and sisters.

Abstract

In this work we deal with algorithms and applications of ranked tree automata, more precisely tree pattern matching and conversion from tree automata to regular tree expressions and vice versa. These fields of language theory have been widely investigated in the string theory but it is not the case for trees. The principle goal of the present thesis is the contribution in the development of the tree language theory by generalizing some string algorithms to trees and providing a toolkit for regular tree languages algorithms and applications.

The first contribution consists on the proposition of a generalization to trees of Thompson's pattern matching algorithm where the pattern set to be matched against is defined by a regular tree expression E. We start by presenting a new method that uses a tree automaton constructed inductively from a regular tree expression E, which is somehow a generalization of Thompson automaton for strings. After that, we run the constructed automaton on the subject tree t. The pattern matching algorithm proposed requires an $O(|t||E|)$ time complexity. The novelty of this contribution besides the low time complexity is that the set of patterns can be infinite, since we use regular tree expressions to represent those patterns.

Another contribution in the tree pattern matching domain is the proposition of a new method treating the fixed tree pattern matching, that is the looking for a single specified pattern in a subject tree. We have adapted the suffix automata and backward string matching algorithms to trees.

Our contribution for the proof of Kleene theorem for tree, besides Thompson tree automaton, has been the proposition of an algorithm converting tree automata to regular tree expressions using integrals, it is also a generalization of the underlying algorithm in strings.

Finally, all the algorithms and methods proposed have been incorporated in our toolkit dedicated to tree automata YATAT.

Keywords : regular tree expression, tree automata, Thompson automata, tree suffixes automata, tree pattern matching, tree automata toolkit.

Résumé

Dans cette thèse, nous nous intéressons aux algorithmes et applications des automates d'arbres rangés, plus particulièrement la recherche de motifs d'arbres et la conversion d'une expression régulière en automate et vice versa. Ces deux aspects de la théorie des langages ont été largement étudiés pour les mots mais ce n'est pas le cas des arbres. Le but principal de cette thèse est la contribution au développement de la théorie des langages d'arbres en proposant des généralisations des algorithmes des mots vers les arbres et aussi de fournir une boîte à outils pour la manipulation des algorithmes des langages d'arbres.

La première contribution consiste en la proposition d'une généralisation de l'algorithme de recherche de motifs de Thompson pour les arbres où le motif recherché est défini par une expression régulière d'arbre. Nous avons d'abords présenté une méthode utilisant un automate construit par induction à partir d'une expression régulière d'arbre E, qui peut être vue comme une généralisation de celle de Thompson pour les mots. Ensuite, nous avons utilisé l'automate construit pour effectuer la recherche de motifs sur l'arbre objet t. La complexité temporelle de l'algorithme de recherche de motifs est de l'ordre de $O(|t||E|)$. L'originalité de cette contribution outre la complexité linéaire, réside dans le fait que l'ensemble des motifs à rechercher peut être infini car nous utilisons les expressions régulières d'arbre pour représenter ces motifs.

Une autre contribution dans le domaine de la recherche de motifs d'arbre est la proposition d'une méthode qui généralise les structures des automates de suffixes des mots vers les arbres, ainsi qu'un algorithme de recherche de motifs généralisant le principe des fenêtres glissantes.

Concernant la preuve du théorème de Kleene pour les arbres, notre contribution hormis la construction de l'automate de Thompson pour les arbres, a été la proposition d'une généralisation aux arbres de la synthèse d'une expression régulière d'arbre d'un automate d'arbre en utilisant les intégrales.

Enfin, la dernière contribution est l'incorporation des algorithmes et méthodes proposés dans une nouvelle boîte à outils pour les automates d'arbres YATAT.

Mots clés : expression régulière d'arbre, automate d'arbre, automate de Thompson, automate des suffixes d'arbres, recherche de motifs d'arbres, boîte à outils d'automates d'arbres.

ملخص

في هذا العمل نتطرق إلى خوارزميات و تطبيقات البنى ذاتية التشغيل و خاصة تطابق النماذج الشجرية و التحويلات من العبارات النموذجية للبنى الشجرية إلى البنى ذاتية التشغيل الشجرية و العكس. هذان المجالان تمت دراستهما بشكل مكثف في مجال نظرية السلاسل و بشكل أقل بالنسبة للبنى الشجرية. الهدف الأساسي لهذه المذكرة هو المساهمة في إثراء نظرية البنى الشجرية عن طريق إقتراح تعميمات لخوارزميات السلاسل إلى البنى الشجرية و توفير أداة برمجية لمعالجة تطبيقات و خوارزميات البنى الشجرية.

المساهمة الأولى في هذه المذكرة هي إقتراح تعميم لخوارزمية ثومبسون من السلاسل إلى البنى الشجرية بهدف إستعمالها في البحث عن نماذج لبنى الشجرية معرفة بعبارة منهجية شجرية. بدأنا بعرض طريقة جديدة تقوم على تركيب بنية شجرية ذاتية التشغيل إستقرائيا إنطلاقا من عبارة منهجية شجرية تمثل النموذج المراد البحث عنه E. هذه الطريقة هي عبارة عن تعميم لطريقة ثومبسون للسلاسل إلى البنى الشجرية. يتم إستعمال هذه البنية الشجرية للبحث عن النموذج E في بنية شجرية t. الخوارزمية المقترحة للبحث عن النماذج تتطلب تعقيد زمني من حوالي $\mathcal{O}(|t||E|)$ حيث $|t|$ تمثل حجم البنية الشجرية و $|E|$ تمثل حجم العبارة المنهجية. أهمية هذه الخوارزمية تكمن ليس فقط في تعقيدها الزمني المنخفض و إنما أيضا في العدد اللامحدود للنماذج التي يمكن البحث عنها في البنى الشجرية نظرا لإستعمالنا العبارات المنهجية لتمثيل النماذج.

المساهمة الأخرى في مجال البحث عن نماذج شجرية هي إقتراح خوارزمية لمشكلة البحث عن بنى شجرية ثابتة. هذه المساهمة هي كذلك تعميم لخوارزمية مشابهة في السلاسل تعتمد على بنى ذاتية التشغيل لجميع عوامل سلسلة ما و تعرف بإسم أوراكل العوامل. كما تم أيضا تعميم مفهوم النوافذ الإنزلاقية في خوارزميات البحث عن نماذج في السلاسل إلى البنى الشجرية.

في هذه المذكرة تطرقنا أيضا إلى برهنة نظرية كلين بالنسبة للبنى الشجرية أي تحويل العبارات المنهجية للبنى الشجريةإلى بنى ذاتية التشغيل الشجرية و العكس. إضافة إلى تعميم خوارزمية ثومبسون السابقة، إقترحنا خوارزمية جديدة لتحويل بنية شجرية ذاتية التشغيل إلى عبارة منهجية بإستعمال التكامل.

أخيرا، كل الخوارزميات التي تم إقتراحها في هذه المذكرة تمت إضافتها إلى الأداة البرمجية الخاصة بالبنى الشجرية.

الكلمات الدلالية: بنية شجرية. بنية ذاتية التشغيل. عبارة منهجية. تحويل بنى ذاتية التشغيل إلى عبارة منهجية و العكس. أدوات برمجية.

Contents

List of Figures

List of Algorithms

List of Abbreviations

DFA Deterministic Finite Automaton

DTA Deterministic Tree Automaton

FCNS First Child Next Sibling

NFA Nondeterministic Finite Automaton

RTE Regular Tree Expression

RTG Regular Tree Grammar

RTL Regular Tree Language

TA Tree Automaton

TPM Tree Pattern Matching

YATAT Yet Another Tree Automata Toolkit

Introduction

Trees are one of the most fundamental ways of structuring data. They are suitable for representing information with hierarchical structure. Trees can be used to represent natural language sentences, computer programs, algebraic formulas, molecule structures, etc.

The formal tree language theory is the study of tree languages that might be defined as subsets of the set of trees over some tree alphabet. This theory also includes the mathematical devices used to describe, recognize and analyze tree languages.

Tree language theory has emerged in the middle of the sixties quite naturally from the view of string automata as unary algebras. The generalization from strings to trees means simply that any finite algebra of finite type can be seen as an automaton which accepts trees over the ranked alphabet as inputs. In this case, strings are considered as trees over a unary ranked alphabet. Hence, the theory of tree languages can be seen as an extension of Büchi's and Wright's one for strings [TW68].

Tree language theory has been applied in different areas: from code generation in compilers (instruction selection or optimization) [AG85, FHP92], term rewriting [Grä91, HO82], to genetics (DNA/RNA pattern matching) [Gie98, DDP+05], XML document processing [LT14], protocol verification (cryptography and network protocols) [GK00, GL00], etc.

Obviously, tree language theory does not deal with subsets of trees as they may get infinite, but rather with a predefined representation of the tree language in question. We distinguish two sorts of representations: one for generating tree languages such as tree grammars, tree expressions, etc. And another one for accepting such languages like tree automata, tree transducers, tree systems, etc.

We say that a tree language is regular if it can be accepted by tree automata, generated by a regular tree grammar or even represented by a regular tree expression [GS15, CDG+08].

Tree automata are one of the regular tree languages aspects that has caught a considerable attention since the late sixties. Generalizing constructions, proving

theorems and proposing new algorithms show how much this field of formal languages is important [CDG+08]. Applications of tree automata include tree acceptance, tree parsing and tree pattern matching problems [Cle08].

Introduced by Kleene [Kle56] for strings, a regular expression is a string formalism allowing the specification of a regular language. It is constructed basically by connecting symbols of the language alphabet with different operators such as union, concatenation and closure. This formalism has been imported to the tree language theory where regular tree expressions specify rather a tree language [RS97].

Objectives

Kleene theorem is considered as a primordial concept in the string's language theory since it proves the equivalence between regular expressions and finite automata by showing that they denote the same regular language. This theorem has been successfully proved for trees [CDG+08]. While conversions from finite automata to regular expressions and back have been extensively studied in strings, it is not the case for trees. In fact, few algorithms have been proposed in order to convert regular tree expressions to tree automata and back. Yet, we still lack studies dealing with these constructions with respect to their properties and complexity for comparison and optimization purposes.

The ultimate aim of the present thesis is the contribution in the development of the tree language theory. This is done through the study of the existing theory and then developing and/or generalizing some algorithms for tree structures. This work has also been motivated by the necessity of providing a toolkit dealing with different formalisms and algorithms of tree languages for experimental reasons.

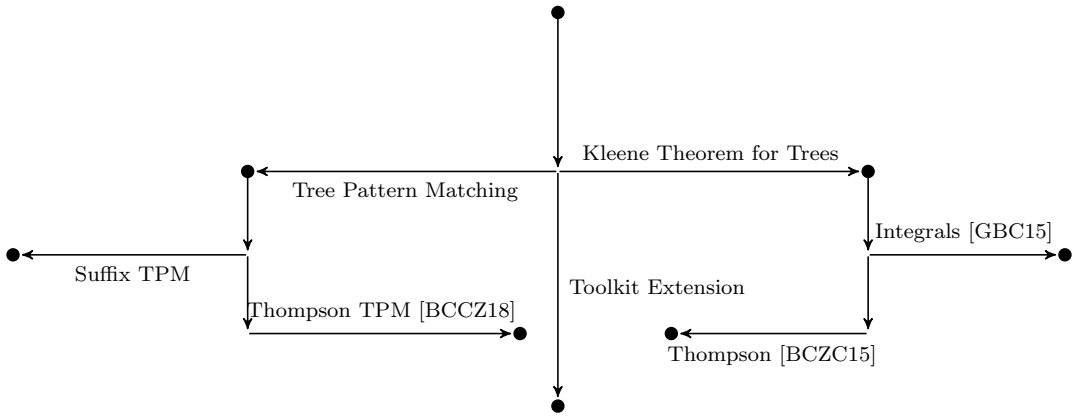


Figure 1: Main contributions

Contributions

In this work we deal with regular tree languages, more precisely with labeled ordered ranked trees. Labeled means that each node of the tree contains a symbol as label. The term ranked implies that the symbol labeling a node determines the number of child nodes of the node. Ordered means that the child nodes of a parent node are ordered, that is: exchanging child nodes or subtrees results in a different tree.

As it is shown in Figure 1, our contributions are in the fields of tree pattern matching and the proof of Kleene theorem for trees.

We have started by proposing a generalization from strings to trees of the well-known Thompson automata [Tho68]. The automata proposed have been used to solve tree pattern matching problem [BCCZ18]. That is to say, looking for one or more occurrences of a tree pattern in a target subject tree.

Another contribution in the tree pattern matching domain has been the proposition of an algorithm for fixed tree pattern matching. This algorithm has been inspired by the factor oracle in strings [CMM99]. We have proposed the construction of a special tree automaton that we called tree suffixes automaton.

For the proof of Kleene theorem, the synthesis of tree automata to regular tree expressions has been treated by introducing the notion of integration on trees [GBC15].

The last contribution of this thesis is the extension of the tree automata toolkit YATAT. Both tree pattern matching algorithms and conversion methods have been incorporated in YATAT. A random generator for regular tree expressions and subject trees has also been added to YATAT for test purposes.

1. Generalization of Thompson's Automata to Trees

In the late sixties, Thompson presented a method that allows the construction of nondeterministic finite string automata inductively from a given regular expression [Tho68]. The resulting automaton is composed of multiple automata fragments constructed depending on the basic operations of the regular expression.

By analogy to strings, we have proposed a construction of a bottom-up tree automata from a regular tree expression that will be used later in tree pattern matching. This is done through the definition of a general form of such automaton that has nearly the same characteristics as Thompson automaton for strings. Indeed, we have designed a special form of tree automaton that allows us to inductively construct a tree automaton from a regular tree expression in a straightforward way. This construction has been used to propose a tree pattern matching algorithm.

2. Tree Pattern Matching Using Thompson Tree Automata

Tree pattern matching algorithms play an important role in many applications such as compilers, validation of XML documents, automatic proofs, etc. This problem can be defined as the search for all occurrences of one or more given trees (called *patterns*) in a subject tree. It also includes testing the occurrences of the pattern, counting their numbers or even determining their positions. Tree pattern matching can be considered as an extension of the string pattern matching problem. Several algorithms have appeared in the literature [FIJ+12, HO82, MS98]. These algorithms are more or less a generalization to trees of the underlying algorithms in the string case.

The most well-known algorithms for pattern matching in strings is the one due to Thompson (1968). It consists of building by induction, an automaton from a regular string expression of a given pattern, then using this automaton to look for the different occurrences of the pattern in a subject text. The original string pattern matching algorithm using Thompson automata for strings has been used in many practical tools such as the *grep* utility on Linux [Goe95, AA04]. Inspired by this algorithm, we have proposed its generalization to trees.

3. Generalization of Integrals to Trees

For the proof of Kleene theorem for trees and besides Thompson generalization to trees which construct a tree automata from a given regular tree expression, another generalization to tree has been proposed in backward. i.e. the synthesis of a regular tree expression from a tree automata using integrals.

Integrals might be considered as the inverse operation of derivatives, so we have also shown the relation between tree integrals and tree derivatives. This generalization is used to denote by a regular tree expression, the accepted tree language of a tree automaton.

4. Tree Pattern Matching Using Tree Suffixes Automata

In tree pattern matching, the tree pattern might be either a fixed subtree or a tree template, i.e., a tree with variable leaves. Fixed string pattern matching has been extensively studied which is not the case for trees. Lately, new finite automata have been introduced such as suffixes/factor oracles in order to optimize the string matching process [CMM99].

Our second contribution in the tree pattern matching field has been the generalization of the idea of the fixed string matching to trees. That is to say, we have proposed a new method similar to the suffix and factor oracles in strings for trees.

5. Tree Automata Toolkit Extension

Many algorithms dedicated to solve problems related to tree automata have been proposed in the literature. However, few toolkits have been implemented in order to manipulate, evaluate or even compare tree automata algorithms [EKM98, GT01, MK06, LSV12].

We have extended our tree automata toolkit YATAT, which stands for *Yet Another Tree Automata Toolkit*, by incorporating all the methods and algorithms proposed in this thesis. The ongoing version of YATAT implements algorithms for construction and manipulation of tree automata and regular tree expressions.

Dissertation Structure

The present thesis is divided in seven chapters apart from this introductory chapter and the conclusion.

Chapter One : Notations and Preliminaries on Words and Trees, contains all mathematical and notational preliminaries necessary for reading the remaining of the thesis.

Chapter Two : Regular Tree Languages gives an insight into the tree language theory, tree automata and regular tree expressions along with their different representations.

These two first chapters might be skipped and referred back to when necessary.

Chapter Three : From Regular Tree Expressions to Tree Automata and Back: State of the Art, is a literature overview of algorithms converting tree automata to regular tree expressions and back with detailed examples and a brief discussion on these conversions.

Chapter Four : Construction of Tree Automata Using Thompson Generalization, presents our first contribution: the Thompson tree automata Construction. We start by giving a recall of Thompson automata for strings and then we introduce our generalization along with proofs of its validity and an example of constructing such automata.

Chapter Five : Synthesis of Regular Tree Expressions Using Integrals, treats the synthesis of a tree automaton into a regular tree expression. We present

here our contribution using the generalization to trees using integrals.

Chapter Six : Tree Pattern Matching, deals with the problem of tree pattern matching and is split in three parts: The first part presents a summarized survey of the tree pattern matching algorithms. While the second part describes a tree pattern matching algorithm using the proposed Thompson tree automaton, the third one proposes another tree pattern matching algorithm using tree suffixes automaton.

Chapter Seven : YATAT: Yet Another Tree Automata Toolkit, is about the presentation and the extension of the tree automata toolkit YATAT.

We conclude by an overview of the problems and algorithms treated in this thesis, and some perspectives for future research.

Chapter 1

Notations and Preliminaries on Words and Trees

In this chapter we give some notations and preliminaries of regular tree languages needed throughout this thesis. After a brief recall of some basic algebraic definitions, we start by the string case in order to get a general idea of the formal languages context. Let us note that in the present manuscript, we use interchangeably the terms *word* and *string*.

1.1 Monoïd and Semirings

A monoïd is a set M together with an associative operation $\odot : M \times M \to M$ and a neutral element 1 in M; i.e. $a \odot 1 = 1 \odot a = a$, for all $a \in M$. If, in addition, $a \odot b = b \odot a$, for all $a, b \in M$, then $(M, \odot, 1)$ is said to be commutative. For example, the set of natural numbers $\mathbb{N}$ together with the multiplication operation forms a monoïd in which 1 is the neutral element.

A semiring is a system $(M, \oplus, \odot, 0, 1)$ where $(M, \oplus, 0)$ is a commutative monoïd and $(M, \odot, 1)$ is a monoïd, in addition to the two following conditions:

- The multiplicative operation distributes over the additive: for every $a, b, c \in M$, it holds that $(a \oplus b) \odot c = (a \odot c) \oplus (b \odot c)$ and that $a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c)$.
- The neutral element with respect to $\odot$ is absorptive: for every $a \in M$, we have that $a \odot 0 = 0 \odot a = 0$.

Examples of semirings would be the set $\mathbb{N}$ (including zero) with ordinary addition and multiplication, and the polynomials over the variable x with natural number coefficients [Hög07].

1.2 Alphabet, Words and Languages

An *alphabet* Σ is a finite nonempty set of elements, each of which is called a symbol or a letter.

A *string* (word) is a finite sequence of symbols from Σ. The empty sequence is called the empty word and noted ε.

The set of all words (resp. of all nonempty words) over an alphabet Σ is denoted Σ^* (resp Σ^+).

The length of a word w noted $|w|$ is the number of its symbols where each symbol is counted as many times as it occurs, that is:

- $|\varepsilon| = 0$,

- For each $\alpha \in \Sigma$, $|\alpha| = 1$,
- For $u, v \in \Sigma^*$, $|uv| = |u| + |v|$.

The concatenation[1] of two words $u, v \in \Sigma^*$ is the operation that creates a new word uv by setting the words u and v side by side. It is an associative but not commutative operation where the empty word ε is the identity: $\varepsilon w = w\varepsilon = w$. The concatenation of n copies of a word $w \in \Sigma^*$ is also denoted by w^n where $w^0 = \varepsilon$.

In algebraic terms, Σ^* and Σ^+ are the free monoïd and semiring generated by Σ with respect to the operation of concatenation and with the neutral element ε.

A formal language over Σ is a finite or infinite subset of Σ^*. An ε-free language is a subset of Σ^+. The empty language is denoted by $\emptyset$.

A finite language is defined by listing all of its words, which is not possible for infinite languages. Other formalisms have been proposed in order to define infinite languages like automata, grammars, and regular expressions.

Example 1.2.1 *Let $\Sigma = \{a, b, c, d\}$. Σ^* is the set of all possibles words constructed using Σ. $\Sigma^* = \{aaa, accd, babbca, dbcabdc, \ldots\}$. Let $\mathcal{L}_1, \mathcal{L}_2 \subset \Sigma^*$ be two languages. $\mathcal{L}_1 = \{\varepsilon, ab, ac, ad\}$ is a finite language. An example of an infinite language might be $\mathcal{L}_2 = (ab)^* = \{\varepsilon, ab, abab, ababab, \ldots\}$.*

Multiple operations can be applied on languages, regarding the latter as sets, such as union, intersection and complementation. We also extend the concatenation from words to languages as follows:

$$\mathcal{L}_1\mathcal{L}_2 = \{w_1 w_2 \mid w_1 \in \mathcal{L}_1 \text{ and } w_2 \in \mathcal{L}_2\}$$

For an integer $n \geq 0$, the n^{th} power of a language $\mathcal{L}$, denoted $\mathcal{L}^n$ is defined as follows:

- $\mathcal{L}^0 = \varepsilon$,
- $\mathcal{L}^n = \mathcal{L}^{n-1}\mathcal{L}$ for $n > 0$.

The concatenation closure or the Kleene star (resp. Kleene plus) noted $\mathcal{L}^*$ (resp. $\mathcal{L}^+$) is the union of all nonnegative powers of $\mathcal{L}$ (resp. of all positive powers of $\mathcal{L}$). That is:

- $\mathcal{L}^* = \bigcup_{i \geq 0} \mathcal{L}^i$.

[1] called also catenation in some references

- $\mathcal{L}^+ = \bigcup_{i>0} \mathcal{L}^i$.

The operations of union, concatenation, and Kleene star are referred to as *regular operations* on formal languages. A language is called regular if it can be obtained from Σ by using the regular operations finitely.

In formal language theory there exist mainly two mechanisms for defining languages: acceptors and generators. For regular languages, the acceptors are finite automata and the generators are regular expressions and regular grammars.

A deterministic finite automaton (DFA) A is a quintuple $(Q, \Sigma, \sigma, s, F)$, where:

- Q is the finite set of states,
- Σ is the input alphabet,
- $\sigma : Q \times \Sigma \to Q$ is the transition function,
- $s \in Q$ is the starting state, and
- $F \subseteq Q$ is the set of final states.

Nondeterministic finite automata are a generalization of DFA where, for a given state and an input symbol, the number of possible transitions can be more than one or having state transitions without reading any input symbol, i.e. ε-transitions.

Formally, a nondeterministic finite automaton A is a quintuple $(Q, \Sigma, \sigma, s, F)$ where:

- Q, Σ, s, and F are defined exactly the same way as for a DFA, and
- $\sigma : Q \times \Sigma \to 2^Q$ (or $\sigma : Q \times \Sigma \cup \{\varepsilon\} \to 2^Q$ for NFA with ε-transitions) is the transition function, where 2^Q denotes the power set of Q.

A regular expression R over an alphabet Σ denoting the language $\mathcal{L}(R)$ is defined inductively as follows:

- $R = \emptyset$, $\mathcal{L}(R) = \emptyset$,
- $R = \varepsilon$, $\mathcal{L}(R) = \{\varepsilon\}$,
- $R = a$ for $a \in \Sigma$, $\mathcal{L}(R) = \{a\}$,
- $R = R_1 + R_2$, $\mathcal{L}(R) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$,
- $R = R_1 R_2$, $\mathcal{L}(R) = \mathcal{L}(R_1)\mathcal{L}(R_2)$,

- $R = (R_1)^*$, $\mathcal{L}(R) = (\mathcal{L}(R_1))^*$.

The complexity of a regular expression is the number of nested stars in the expression, called the star height of the expression. It's considered as an important metric for major problems in formal language theory [GH15].

There are many ways to decide whether a language is regular or not; for example, this can be done by demonstrating that the language is accepted by a finite automaton, could be specified by a regular expression, or generated by a regular grammar.

On the other hand, to prove that a language is not regular, the most commonly used tools are the pumping properties of regular languages, which are usually stated as *pumping lemmas*. The term *pumping* intuitively describes the property that any sufficiently long word of the language has a nonempty subword that can be "pumped", which means that if a subword is replaced by an arbitrary number of copies of the same subword, the resulting word is still in the language [RS97].

A regular grammar is a quadruple $G = (N, T, S, P)$, where:

- N and T are disjoint alphabets called nonterminal and terminal symbols respectively,
- $S \in N$ is the start symbol or the axiom,
- $P : \Sigma_G^* N \Sigma_G^* \times \Sigma_G^*$ is the set of production rules with $\Sigma_G = N \cup T$.

$(u, v) \in P$ is noted $u \to v$.

1.3 Trees: Definitions

A tree is a set of nodes organized in hierarchical structure. Trees arise everywhere in computer science, and there are numerous formalisms in the literature for describing and manipulating them. Some of these formalisms are declarative and based on logical specifications: for example, first-order logic, monadic second-order logic, ... Others are procedural formalisms such as tree automata, tree transducers, or tree grammars. These formalisms have found multiple applications in verification, program analysis, logic programming, constraint programming, linguistics, and databases. Examples of tree structures include file systems, games (particularly board games), XML (see for example 1.1), ... ([Knu97, CLRS09]).

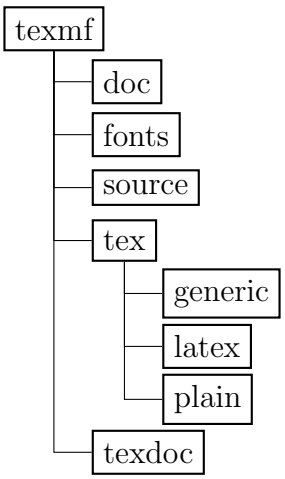


Figure 1.1: Example of a tree file system

Each node in a tree has zero or more child nodes. A node that has a child is called the child's parent node (or ancestor). A node has at most one parent. The topmost node in a tree is called the *root*.

As described in the introduction, this thesis focuses on the domain of regular tree languages on ranked trees, to be more precise rooted, ranked, ordered, node labeled trees.

- A *rooted* tree is a tree, where one node is marked as root node . Any tree can be turned into rooted tree by marking one of its nodes as root node. The root node is the main node of a tree, its direct neighbors are its children which can have their own children and so on.
- *Labeled* node means that each node in a tree contains a symbol as label.
- The term *ranked* implies that the symbol labeling a node determines the number of child nodes of the node. It is not allowed for a symbol to appear in nodes with different numbers of child nodes.
- *Orderedness* addresses the fact that the child nodes of a parent node are ordered. A node with a symbol of rank n has n child nodes, where the leftmost child is considered the first child and the rightmost child the n^{th} child. Exchanging child nodes/subtrees results in a different tree if the exchanged items are not equal.

These descriptions of trees are quite informal. There are mainly two ways to define trees formally. The first one called tree domain representation, which uses a decimal notation for identifying the nodes of a tree [CDG+08]. In the other one,

trees are defined as terms, i.e. syntactic objects. These two representations are given hereafter when defining ranked and unranked trees.

1.3.1 Ranked vs Unranked Trees

Until recently, most algorithms and applications for trees dealt with ranked trees. In such trees, all nodes with the same label have the same rank, i.e. the number of children of a node is determined by the label of that node. However, recently the focus has shifted towards unranked trees, in which there are no restrictions on the number of children a node can have. This systematic study has been initiated by the development of XML, a data format which has become a standard for information exchange on the World Wide Web. In particular, XML data is typically modeled as unranked labeled trees [Lib06].

Definition 1.3.1 *Let* (Σ, ar) *be* a ranked alphabet*, where* Σ *is a finite set of symbols and* ar *represents the* rank *of* Σ *which is a mapping from* Σ *into* $\mathbb{N}$*.*

The set of symbols of rank n *is denoted by* Σ_n*. The elements of rank* 0 *are called* constants *or* leaves *.*

Definition 1.3.2 *A ranked* tree t *over* Σ *is inductively defined as follows:*

- $t = a$,
- $t = f(t_1, \ldots, t_k)$

where a *is a constant,* k *is any integer satisfying* $k \geq 1$*,* f *is any symbol in* Σ_k *and* $t_1, \ldots, t_k$ *are any* k *trees over* Σ*.*

Unranked trees are defined in a similar way as ranked ones but over a simple alphabet where the symbols do not have arities. Let Σ be an unranked alphabet, i.e. just a finite set of symbols (See Figure 1.2 for example). All the definitions from ranked trees, such as nodes, leaves, etc. can be immediately transferred to unranked trees. Finite sequences of unranked trees are called hedges [Cou78].

Example 1.3.3 *Let* $A = \{(f,2),(g,1),(a,0),(b,0)\}$ *be a ranked alphabet.* $t_1 = f(g(a),b)$ *is a ranked tree. An example of an unranked tree would be* $t_2 = a(b(b,c),c(b,b(a)))$ *from the unranked alphabet* $A' = \{a,b,c\}$*.*

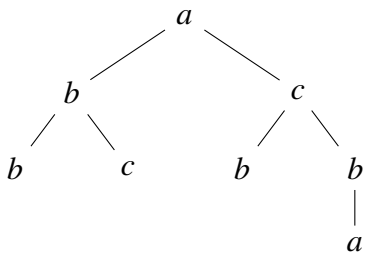


Figure 1.2: Example of an unranked tree.

Many definitions that have been given for ranked trees and that do not make use of the ranked nature of the tree, such as height or subtree, directly carry over to unranked trees [CDG+08].

Since unranked trees are unbounded vertically and horizontally, it is often useful to encode them by ranked trees. In particular, when dealing with tree automata, a lot of results for ranked tree automata can be generalized to the unranked trees using such encodings.

The *First Child Next Sibling* encoding (FCNS) is a very natural encoding when unranked trees have to be represented as data structures using pointers. For each node we have one pointer to its first (left-most) child, and one pointer to its next right sibling. All symbols from the unranked alphabet become binary symbols in the ranked alphabet, and a new symbol $\perp$ of arity 0 is added. The first child of a node in the FCNS encoding corresponds to its first child in the unranked tree, and the second child of a node in the FCNS encoding corresponds to its right sibling in the unranked tree. If the corresponding nodes (first child or right sibling) do not exist in the unranked tree, then we put the symbol $\perp$ [Löd12].

Note that in this way we represent trees as words (with a special structure) over the alphabet $\Sigma \cup \{,\} \cup \{(,)\}$. An alternative way of representing trees which makes their hierarchical structure even more explicit is the tree domain representation.

Definition 1.3.4 *The domain $\mathbb{D}(t) \subseteq \mathbb{N}$ of a tree $t = f(t_1,\ldots,t_{ar(f)})$ is defined inductively as: $\mathbb{D}(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{ar(f)} i.\,\mathbb{D}(t_i)$, where the . denotes the concatenation of words. The elements of the domain $\mathbb{D}$ are called the nodes of the tree.*

Tree domains were introduced by S. Gorn in 1965 [Bra69]. It describes the paths to all nodes and thereby describes the basic structure of a tree. Each tree domain $\mathbb{D}$ may be regarded as an unlabeled tree where its nodes correspond to the elements of $\mathbb{D}$. The root of the tree is represented by ε, and the leaves are the nodes which are maximal with respect to the prefix relation.

Example 1.3.5 $\mathbb{D} = \{\varepsilon, 1, 2, 1.1, 1.2, 2.1\}$ *is a tree domain which is depicted as the unlabeled tree of Figure 1.3. The set* $\mathbb{D}$ *is called the domain of the tree* t.

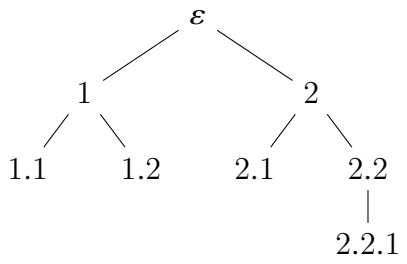


Figure 1.3: Example of a tree domain representation.

Since these two formalisms are equivalent, choosing between them is a matter of convenience. We use mainly the definition of trees as terms which is more suitable for algebraic treatments.

1.3.2 Ordered vs Unordered Trees

An ordered tree means that the child nodes of a parent node are ordered. A node with a symbol of rank n has n child nodes, where the leftmost child is considered as the first child and the rightmost child the n^{th} child. Exchanging different child nodes results in a different tree.

Example 1.3.6 *Let* $A = \{(f,2),(g,1),(a,0),(b,0)\}$ *be a ranked alphabet. Let* $t_1 = f(g(a), b)$ *and* $t_2 = f(b, g(a))$. t_1 *and* t_2 *are equivalent unordered tree, but different ordered trees.*

1.3.3 Common Operations on Trees

Definition 1.3.7 *The height of a tree is the length of the longest downward path to a leaf from the root, it is defined inductively as follows:*

- $Height(a) = 0$ *for* $a \in \Sigma_0$ *and*
- $Height(f(t_1,\ldots,t_{ar(f)})) = MAX(Height(t_i)) + 1$, *with* $i = 1,\ldots,ar(f)$.

Example 1.3.8 *The height of the tree* t *in Figure 1.3 equals* 3.

Definition 1.3.9 *The size of a tree* t *denoted by* $|t|$ *is defined inductively as follows:*

- $Size(a) = 1$ *for* $a \in \Sigma_0$, *and*

- $Size(f(t_1,\ldots,t_{ar(f)})) = 1 + \sum_{i=1}^{ar(f)} Size(t_i)$.

For the tree domain representation $|t| = Card(\mathbb{D})$.

Example 1.3.10 *The size of the tree t in Figure 1.3 equals* 8.

Definition 1.3.11 *A subtree of a tree t in a node n denoted by t/n is the tree consisting of n and all its descendants. Note that $t/\varepsilon = t$.*

A proper subtree s of a tree t is a subtree different from t.

Example 1.3.12 *Let $t = a(f(g(c),f(b,b)),h(c))$ be a tree. $f(g(c), f(b,b))$, $g(c)$, $f(b,b)$, c, ... are proper subtrees of t (Figure 1.4).*

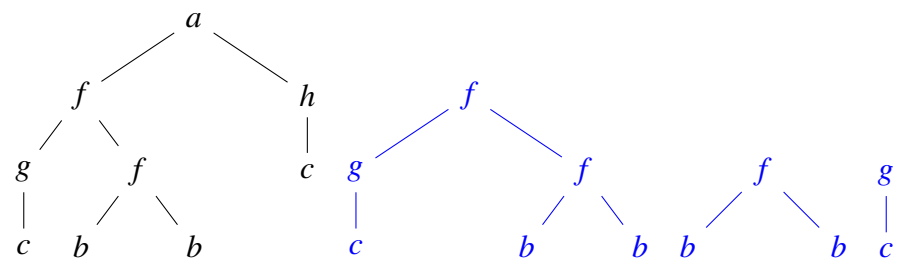


Figure 1.4: Example of a tree (the leftmost tree) and some of its proper subtrees.

Definition 1.3.13 *A string-path $SP(t)$ of a tree t is the set of all strings representing t. It is constructed by concatenating the successive node labels of all paths from the root to all leaves as follows:*

$$SP(t) = \begin{cases} t & \text{if } t \in \Sigma_0 \\ \bigcup\limits_{1\le i\le ar(t)} i.SP(t/i) & \text{otherwise.} \end{cases}$$

Note that t is uniquely defined by the set of its string-paths.

Example 1.3.14 *Let a tree $t = f(f(a,b),g(b))$.*

$SP(t) = \{f1f1a, f1f2b, f2g1b\}$ is the set of string-paths of t.

Definition 1.3.15 *Tree substitution in a tree t is the replacement of a node a (or a subtree t/a) by another subtree s. There exist different forms of tree substitution [Cle08]:*

- *Substitution of a single subtree occurrence, indicated by its root node, by another subtree.*

- *Substitution of all occurrences of a subtree by occurrences of another subtree.*
- *Concurrent substitution at multiple leaf symbols, where all occurrences of the same leaf symbol are replaced by the same subtree. This operation is also called tree concatenation and denoted $t._a s$, particularly, the restriction to a single leaf symbol is called tree product.*
- *Substitution of occurrences of a leaf symbol by different subtrees, where all occurrences of a single leaf symbol are replaced by different subtrees.*

Example 1.3.16 *Let $t = f(f(a,a),h(c))$.*

$t[a_1 \leftarrow f(a,b)] = f(f(f(a,b),a),h(c))$ is an example of substitution of the first form.

$t[a \leftarrow f(a,b)] = f(f(f(a,b),f(a,b)),h(c))$ is an example of substitution of the second and third form.

$t[a_1 \leftarrow g(c), a_2 \leftarrow h(b)] = f(f(g(c),h(b)),h(c))$ is an example of substitution of the fourth form.

1.3.4 Weighted Trees

Definition 1.3.17 *A weighted tree is a tree whose nodes and/or edges are assigned a weight, generally an integer number.*

This definition will be further developed in the next chapter when we present weighted tree automata and tree series.

Example 1.3.18 *Figure 1.5 is an example of a weighted tree. Here we assigned a weight to each node.*

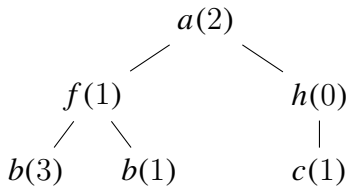


Figure 1.5: Example of a weighted tree.

1.4 Conclusion

In this chapter, we have briefly given some notations and preliminaries about words and trees. We have focused on labeled ordered ranked trees as they will be used later to present the tree language theory state of the art and then to propose new algorithms for proving Kleene theorem for trees and tree pattern matching problem. Since trees are considered more or less a generalization of words, let us note how words, can be viewed as trees where each symbol has rank 1. For example, the word abc becomes the tree $a(b(c(\varepsilon)))$. This observation provides useful guidelines for generalizing string algorithms and approaches to tree.

Chapter 2

Regular Tree Languages

In this chapter we present the basic notions of the regular tree language theory along with its different representations particularly tree automata and regular tree expressions.

2.1 Tree Languages

The concept of tree language is comparable to the one of string language, it is defined as a set of trees. The importance of tree language resides in its ability to describe subsets of all possible trees, which facilitate problem descriptions like for example: *"Given a tree t and a tree language $\mathcal{L}$, determine whether $t \in \mathcal{L}$".*

Definition 2.1.1 *Let T_Σ be the set of all trees over an alphabet Σ. A* tree language *is a subset of T_Σ.*

Usually tree languages are not represented by a set of trees, but rather defined indirectly by a special formalism viz. a tree expression, a tree automaton or a tree grammar.

Before giving the formal definition of a tree language, we have to define the concatenation operation on tree languages as an extension of the definition of word concatenation given in the previous chapter. We first give a definition of substituting tree languages in a single tree, and then use it to define the substitution of tree languages in another tree language.

Definition 2.1.2 *Let $a \in \Sigma_n$ and $\mathcal{L}_1,\ldots,\mathcal{L}_n \in T_\Sigma$, then we can define a new tree $a(\mathcal{L}_1,\ldots,\mathcal{L}_n)$ such that $a(\mathcal{L}_1,\ldots,\mathcal{L}_n) = a(t_1,t_2,\ldots,t_n)$ where $t_1 \in \mathcal{L}_1,\ldots,t_n \in \mathcal{L}_n$.*

Definition 2.1.3 *Let $t \in T_\Sigma$, $a_1,a_2,\ldots,a_n \in \Sigma_0$ and $\mathcal{L}_1,\mathcal{L}_2,\ldots,\mathcal{L}_n \in T_\Sigma$. The substitution (or concatenation) in a tree t in $a_1,a_2,\ldots,a_n$ by $\mathcal{L}_1,\ldots,\mathcal{L}_n$, denoted $t(a_1 \leftarrow \mathcal{L}_1, a_2 \leftarrow \mathcal{L}_2,\ldots,a_n \leftarrow \mathcal{L}_n)$, is a tree language defined as follows:*

1. *$\mathcal{L}_i$ if $t = a_i$ with $1 \le i \le n$,*
2. *$\{c\}$ if $t = c \in \Sigma_0$ and $c \neq a_i$ for all $i, 1 \le i \le n$,*
3. *$f(t_1(a_1 \leftarrow \mathcal{L}_1,\ldots,a_n \leftarrow \mathcal{L}_n),\ldots,t_n(a_1 \leftarrow \mathcal{L}_1,\ldots,a_n \leftarrow \mathcal{L}_n))$ if $t = f(t_1,t_2,\ldots,t_n)$ with $f \in \Sigma_n$ and $t_1,t_2,\ldots,t_n \in T_\Sigma$.*

Definition 2.1.4 *Let $t \in T_\Sigma$, $a_1,a_2,\ldots,a_n \in \Sigma_0$ and $\mathcal{L}_1,\mathcal{L}_2,\ldots,\mathcal{L}_n \in T_\Sigma$. The language substitution or language concatenation in $\mathcal{L}$ of $a_1,\ldots,a_n$ by $\mathcal{L}_1,\ldots,\mathcal{L}_n$ denoted $\mathcal{L}(a_1 \leftarrow \mathcal{L}_1,\ldots,a_n \leftarrow \mathcal{L}_n)$ is defined as follows:*

$$\bigcup_{t\in\mathcal{L}} t(a_1 \leftarrow \mathcal{L}_1,\ldots,a_n \leftarrow \mathcal{L}_n)$$

We denote the language substitution (or concatenation) $\mathcal{L}(a \leftarrow \mathcal{L}')$ by $\mathcal{L}._a\mathcal{L}'$, with $a \in \Sigma_0$ and $\mathcal{L}, \mathcal{L}' \in T_\Sigma$.

Definition 2.1.5 *Let $\mathcal{L} \in T_\Sigma$ and $a \in \Sigma_0$, then the language exponentiation is defined as follows:*

- $\mathcal{L}^{0_a} = \{a\}$,
- $\mathcal{L}^{(i+1)_a} = \mathcal{L}^{i_a}._a(\mathcal{L} \cup \{a\})$ *for* $i \geq 0$.

Definition 2.1.6 *Let $\mathcal{L} \in T_\Sigma$ and $a \in \Sigma_0$, then the language closure is defined as follows:*

$$\mathcal{L}^{*_a} = \bigcup_{i \geq 0} \mathcal{L}^{i_a}$$

Example 2.1.7 *Let $\mathcal{L} = \{f(a,b), h(c), c\}$ be a language.*

$\mathcal{L}^{0_c} = \{c\}$

$\mathcal{L}^{1_c} = \{c, h(c), f(a,b)\}$

$\mathcal{L}^{2_c} = \{c, h(c), f(a,b), h(h(c)), h(f(a,b))\}$

$\ldots$

$\mathcal{L}^{*_c} = \{c, h(c), f(a,b), h(h(c)), h(f(a,b)), \ldots, h(h(h \ldots (c))), h(h(h \ldots (f(a,b))))\}$

2.2 Regular Tree Languages

The set of regular tree languages over a ranked alphabet (Σ, ar), denoted $RTL(\Sigma, ar)$, is defined as the smallest set such that:

1. $\emptyset \in RTL$,
2. $a \in RTL$ for all $a \in \Sigma_0$,
3. $f(\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n) \in RTL$ for each $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n \in RTL$ and $f \in \Sigma_n$,
4. $\mathcal{L}_1 \cup \mathcal{L}_2 \in RTL$ for $\mathcal{L}_1, \mathcal{L}_2 \in RTL$,
5. $\mathcal{L}_1._a\mathcal{L}_2 \in RTL$ for $\mathcal{L}_1, \mathcal{L}_2 \in RTL$ and $a \in \Sigma_0$,
6. $\mathcal{L}^{*_a} \in RTL$ for $\mathcal{L} \in RTL$ and $a \in \Sigma_0$.

2.2.1 Pumping Lemma

Like in the word case, the pumping lemma is useful in proving that a tree language is not recognizable by a tree automata and also for solving problems of emptiness and finiteness of tree languages [RS97].

Definition 2.2.1 *Let $\mathcal{L}$ be a tree language. Then, there exists a constant $k > 0$ such that for each tree $t \in \mathcal{L}$ with $Height(t) > k$, t can be written in the form $t_0(t_1(t_2))$, where:*

- $t_1 \neq x_1$ *(x_1 is the first child of t_1),*
- $Height(t_1(t_2)) \leq k$,
- $t_0(t_1^n(t_2)) \in \mathcal{L}$ *for all* $n \geq 0$.

In other words, if $\mathcal{L}$ is a regular tree language, then every sufficiently large tree $t \in \mathcal{L}$ can be decomposed into $t_0(t_1(t_2))$ in such a way that $t_0(t_1^n(t_2)) \in \mathcal{L}$, for all $n \in N$ [DH03].

2.2.2 Closure Properties of Regular Tree Languages

One of the most important results in string language theory that has been verified in the tree language one is the closure properties under a set of operations. That is to say, applying these operations on a regular tree language, the result is also a regular tree language.

Theorem 2.2.2 *The class of regular tree languages is closed under union, intersection, and complement [Eng15].*

For the closure properties of regular tree languages it has been observed that *complete* deterministic tree automata can be complemented by exchanging final and non-final states. For the *intersection*, we might apply a standard product construction simulating both automata, and accepting if both of them reach a final state. For the *union*, we simply take the disjoint union of the components of two nondeterministic tree automata. In case we want to take the union of two deterministic automata and want to obtain a deterministic one, we also use a product construction [Löd12, CDG+08].

Hereafter, we present the three most important formalisms of regular tree language, viz. tree automata, regular tree expressions, regular grammars.

2.3 Tree Automata

In string language theory, word automata play a significant role in many applications and algorithms. The generalization to tree automata is considered as a powerful formalism to represent regular tree languages and the underlying algorithms. Like in the word case, There exist multiple types of tree automata: deterministic vs nondeterministic, ε-free vs automata with ε-transitions, ...

Nevertheless, this generalization has implied the appearance of new properties of tree automata such as the direction in which trees are treated: top-down vs bottom-up automata. Top-down, called also root-to-frontier processing starts at the top, i.e. from the root of the tree so the root states are used as start states and the leaf ones as final states. The other processing type is the bottom-up or frontier-to-root automata. They start from leaves to the root, where the starting states are the leaves and the accepting state is the root one.

Definition 2.3.1 *A tree automaton $\mathcal{A}$ is a quadruple (Q,Σ,Δ,Q_T) such that*

- *Q is a finite state set,*
- *Σ is a ranked alphabet,*
- *Δ is the set of transition rules defined from $Q \times Q^n$ (or $Q^n \times Q$),*
- *$Q_T \in Q$ is the set of final states (root or leaf accepting).*

We denote by Δ^* the function from T_Σ to 2^Q defined for any tree in T_Σ as follows:

- $\Delta^*(t) = \Delta(a)$ if $t = a$ with $a \in \Sigma_0$,
- $\Delta^*(t) = \Delta(f,\Delta^*(t_1),\ldots,\Delta^*(t_n))$ $\Delta^*(t) = \Delta(\Delta^*(t_1),\ldots,\Delta^*(t_n),f)$ if $t = f(t_1,\ldots,t_n)$,

with $f \in \Sigma_n$ and $t_1,\ldots,t_n \in T_\Sigma$.

Definition 2.3.2 *A tree t is accepted by a tree automaton $\mathcal{A}$ iff $\Delta^*(t) \cap Q_T \neq \emptyset$.*

Definition 2.3.3 *The language $\mathcal{L}(\mathcal{A})$ recognized by $\mathcal{A}$ is the set of trees accepted by $\mathcal{A}$, that is $\mathcal{L}(\mathcal{A}) = \{t \in T_\Sigma \mid \Delta^*(t) \cap Q_T \neq \emptyset\}$.*

Strings can be read forwards or backwards. The direction in which a string is read does not affect the power of the automaton, although it can affect its size. In trees case this is true for the nondeterministic case but not in the deterministic one [Dre09] as we will see later, where it is shown that deterministic top-down tree automata are much weaker.

2.3.1 Top-down Tree Automata

In this type of automata, transition rules are directed from the root to the leaves. The processing of trees starts from root and moves downward, which explains the other denomination *root-to-frontier*.

Definition 2.3.4 *A top-down tree automaton $\mathcal{A}$ is a tuple (Q, Σ, Q_T, Δ) where:*

- *Q is a finite set of states,*
- *$Q_T \subseteq Q$ is the set of leaf accepting states,*
- *and Δ is the set of transition rules defined from $Q \times \Sigma_n$ to 2^Q. i.e. $(q_1, \ldots, q_n, f, q) \in \Delta \Leftrightarrow (q_1, \ldots, q_n) \in \Delta(q, f)$.*

2.3.2 Bottom-up Tree Automata

As it is clear from its name, in bottom-up or frontier-to-root automaton transition rules are directed from leaves to the root. The processing of trees starts from leaves and moves upward.

Definition 2.3.5 *A bottom-up finite tree automaton $\mathcal{A}$ is a tuple (Q, Σ, Q_T, Δ) where*

- *Q is a finite set of states,*
- *$Q_T \subseteq Q$ is the set of root accepting states,*
- *and Δ is the set of transition rules defined from $Q^n \times \Sigma_n$ to 2^Q. i.e. $(q, f, q_1, \ldots, q_n) \in \Delta \Leftrightarrow q \in \Delta(q_1, \ldots, q_n, f)$.*

Unlike with the string case where processing left-to-right or right-to-left does not matter, top-down tree automata are no longer equivalent to the bottom-up ones. In particular, top-down deterministic tree automata are much less expressive than their bottom-up counterparts. Furthermore, deterministic top-down tree automata do not verify many of the important closure properties for tree languages. For instance, they are not closed under union and complement [MNS08].

2.3.3 Graphical Representation of a Tree Automaton

The graphical representation of a tree automaton is similar to the one of strings. The states are represented by circles with double circles for the final states, and transitions between states by edges labeled with a symbol of the alphabet Σ_0 or ε.

For tree automata, some changes are made in the representation of transitions [Cle08]. A transition from state q_0 to states $q_1, q_2, ..., q_n$ with a symbol or an ε-transition is represented by:

i. an edge connecting the state q_0 to a small unlabeled circle with a symbol or ε, and;

ii. n edges connecting the small circle to states q_i labeled by i where $i : 1..n$.

In the case of directed automata (top-down or bottom- up) edges are directed (see for example Figures 2.1, 2.2, 2.3).

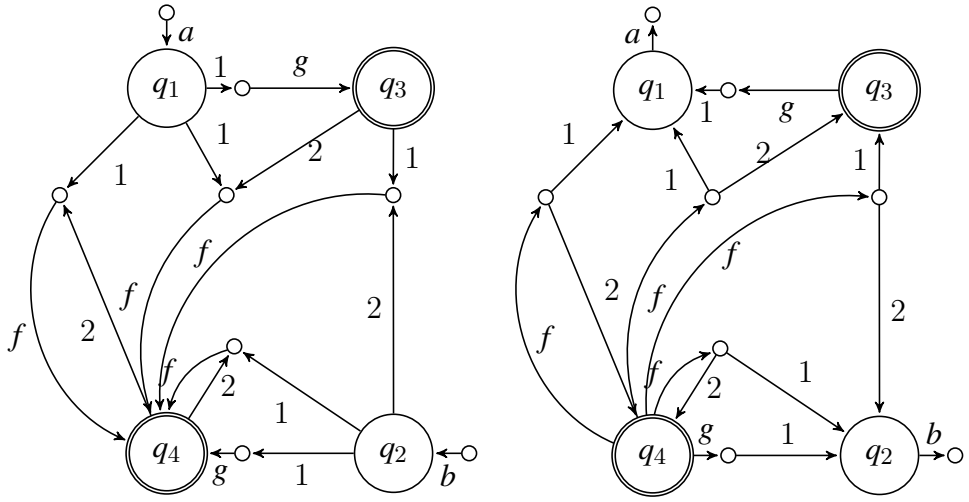


Figure 2.1: Examples of top-down and bottom-up tree automata [Gue17].

2.3.4 Deterministic vs Nondeterministic Tree Automata

Nondeterminism can be described in a similar way as for string automata. We say that a tree automaton is nondeterministic if there exist more than one transition rule for a symbol $a \in \Sigma$ or an ε-transition from any state q in a top-down automaton or a vector of states $(q_1, \ldots, q_n)$ for a bottom-up tree automaton (see Figure 2.2).

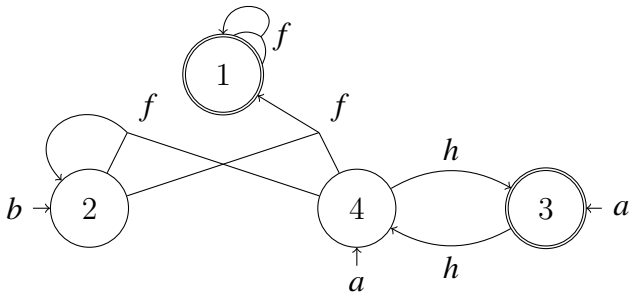


Figure 2.2: Example of a nondeterministic tree automaton.

Consequently, in a deterministic tree automaton there exist neither ε-transitions nor more than one transition for a symbol $a \in \Sigma$ from any state q in a top-down automaton or a vector of states $(q_1, \ldots, q_n)$ for a bottom-up tree automaton. Thus, a deterministic tree automata is unambiguous (see Figure 2.3).

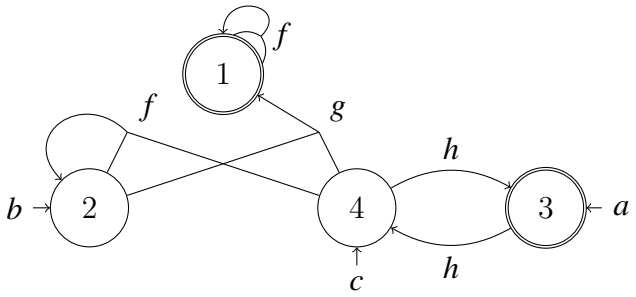


Figure 2.3: Example of a deterministic tree automaton.

Theorem 2.3.6 *For every nondeterministic tree automaton, there exist an equivalent deterministic tree automaton. In the worst case, the number of states of the deterministic automaton is exponential in the number of states of the given nondeterministic automaton.*

The proof of this theorem is straightforward based on the word case theorem. It can be found in [Löd12].

Definition 2.3.7 *We say that a tree automaton $\mathcal{A}$ is a complete automaton if for each state $q \in Q$, there exists at least a transition rule $f(q_1, \ldots, q_n) \to q$ for top-down automaton or $q \to f(q_1, \ldots, q_n)$ for the bottom-up one. q is called an accessible state.*

Definition 2.3.8 *A tree automaton $\mathcal{A}$ is reduced iff for each $q \in Q$, q is accessible.*

The proof of the following theorems can be found in the reference book of tree automata [CDG+08]. We call a recognizable tree language, the language that is recognized by a finite tree automata.

Theorem 2.3.9 *Let $\mathcal{L}$ be a recognizable tree language. Then, there exists a complete tree automaton that accepts $\mathcal{L}$.*

Theorem 2.3.10 *Let $\mathcal{L}$ be a recognizable tree language. Then, there exists a reduced tree automaton that accepts $\mathcal{L}$.*

Theorem 2.3.11 *Let $\mathcal{L}$ be a recognizable tree language. Then there exists a deterministic tree automaton that accepts $\mathcal{L}$.*

It is easy to show that nondeterministic top-down tree automata recognize the same class of tree languages as the nondeterministic bottom-up ones and, thus, deterministic bottom-up tree automata. The following theorem states that deterministic top-down tree automata are not as powerful as these three classes, by showing that they do not even recognize all finite tree languages. This is done by showing that every finite tree language $\mathcal{L}$ can be recognized by a deterministic bottom-up whose states correspond to the subtrees of the trees in $\mathcal{L}$.

Theorem 2.3.12 *There exist tree languages that are recognizable by deterministic bottom-up tree automata but not by any deterministic top-down tree automata.*

This theorem is proven using a counterexample.

Proof. Let $\Sigma = \{(a,0),(b,0),(f,1)\}$ be a ranked alphabet and the finite tree language $\mathcal{L} = \{f(a,b), f(b,a)\}$. Suppose that a deterministic top-down tree automata $\mathcal{A} = (Q,\Sigma,\Delta,Q_T)$ recognizes $\mathcal{L}$. Since $f(a,b) \in \mathcal{L}(\mathcal{A})$, we must have $q_1 \in \Delta_a^*$ and $q_2 \in \Delta_b^*$. But, since $f(b,a) \in \mathcal{L}(\mathcal{A})$, we also have $q_1 \in \Delta_b^*$ and $q_2 \in \Delta_a^*$. This means that $Q_T \in \Delta^*(f(a,a))$ (and $Q_T \in \Delta^*(f(b,b))$), which shows that the deterministic top-down automaton $\mathcal{A}$ accepts $f(a,a)$ and $f(b,b)$ which contradicts the assumption that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$. □

Deterministic and nondeterministic bottom-up tree automata were defined, and their equivalence was established, by Thatcher and Wright [TW68], Doner [Don71] and Magidor and Moran [MAMO69]. Top-down tree automata were introduced by Rabin [Rab68], and Magidor and Moran [MAMO69].

2.3.5 Complexity and Decision Problems of Tree Automata

For estimating the complexity of algorithms dealing with tree automata we have to define the size of a tree automaton. A natural and common way of doing this is to take the number of states and for each transition the number of its entries.

Definition 2.3.13 *Let $\mathcal{A} = (Q,\Sigma,\Delta,F)$ be a nondeterministic tree automaton. The size of a transition $\delta = (f,q_1,\ldots,q_n,q) \in \Delta$ is $|\delta| = n+2$, and the size of $\mathcal{A}$ is $|\mathcal{A}| = |Q| + \sum_{\delta\in\Delta} |\delta|$.*

Hereafter, we enumerate the most important decision problems:

1. The membership problem: for a given nondeterministic tree automaton $\mathcal{A}$ and a tree t, decide whether $t \in \mathcal{L}$ (A).

2. The emptiness problem: for a given nondeterministic tree automaton $\mathcal{A}$, tell if $\mathcal{L}(\mathcal{A}) = \emptyset$.

3. The universality problem: for a given nondeterministic tree automaton $\mathcal{A}$ verify if $\mathcal{L}(\mathcal{A}) = T_\Sigma$.

4. The inclusion problem, for a given nondeterministic tree automata $\mathcal{A}_1$ and $\mathcal{A}_2$, is $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$?

5. The equivalence problem, for a given nondeterministic tree automata $\mathcal{A}_1$ and $\mathcal{A}_2$, do $\mathcal{A}_1$ and $\mathcal{A}_2$ recognize the same language.

The proofs of the following theorems can be found in [Löd12, MNS08, CDG+08, Eng15].

Theorem 2.3.14 *The membership problem can be solved in polynomial time.*

Theorem 2.3.15 *The emptiness problem for a nondeterministic tree automaton can be solved in linear time.*

Theorem 2.3.16 *The universality problem for a tree automata is Exptime-complete.*

Theorem 2.3.17 *The inclusion and the equivalence problem for a tree automata are Exptime-complete.*

2.3.6 Minimization of Tree Automata

The tree automata minimization is also a generalization of the word automata minimization. It is based on the identification of equivalent states of the automaton and merges them in order to reduce the size of the automaton without altering the recognized language. Minimization is of a major interest as it allows the reduction of memory space that a tree automaton takes, and thus the processing time [Hop08]. There exist two major approaches of minimization in the literature: while the most known ones are about the reduction of tree automata states, the other methods deal rather with the number of transitions reduction. For an example of minimization see Example 2.4.

In his PhD thesis [Gue17], Guellouma has classified and discussed the most important algorithms of tree automata minimization for both exact and approached methods on ranked/unranked trees.

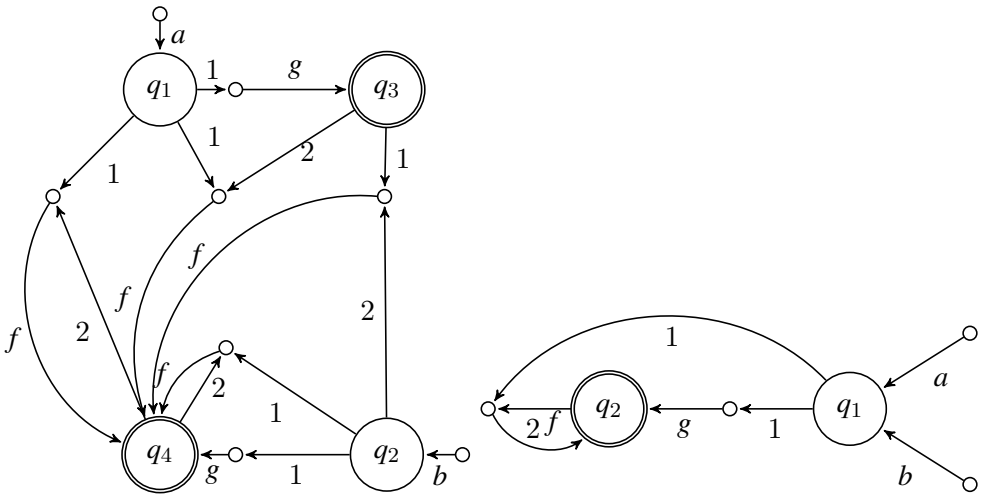


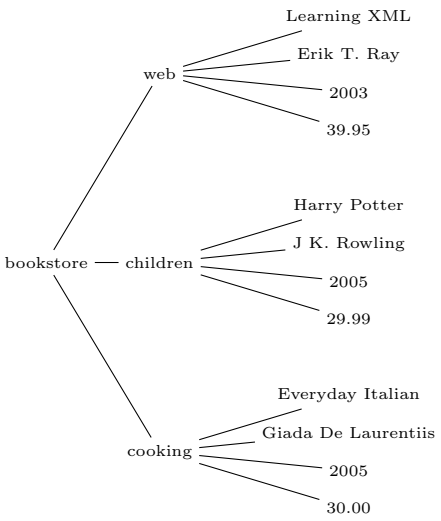
Figure 2.4: Example of a tree automata (left automaton) and its minimized version (right automaton) [Gue17].

2.3.7 Other Tree Automata

In this section, we briefly present other types of tree automata: unranked tree automata, weighted tree automata, and pushdown tree automata.

Unranked or Hedge Tree Automata

A major motivation for looking at unranked trees is the eXtensible Markup Language (XML) since an XML document can be viewed as a tree (see for example Figure 2.5). Consequently, an XML document class corresponds to a tree language [Dre09]. A hedge is a sequence of unranked trees whose properties has been extended from the tree language theory. Note that ordinary trees are hedges that have exactly one root [BKMW01].




```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

Figure 2.5: Example of an XML document and its tree representation.

Definition 2.3.18 *A hedge $\mathcal{H}$ over an alphabet Σ is defined inductively as follows:*

- *The empty term sequence denoted by ε is a hedge,*
- *the symbol $a \in \Sigma$ is a hedge,*
- *if g is a hedge and $a \in \Sigma$, then $a(g)$ is a hedge,*
- *if g and h are hedges, then gh is a hedge.*

Example 2.3.19 *The term $a(a(a()a())a(a()a()))$ represents a complete binary tree of height 3, all of whose nodes have the label a.*

Hereafter, we give the formal definition of a deterministic hedge automaton [Mur99, Lib06]. The transitions are modeled by a word language. These languages occurring in the transition rules are called horizontal languages [CDG+08].

Definition 2.3.20 *A deterministic hedge automaton over an alphabet* Σ *is the tuple* $\mathcal{H} = (\Sigma, Q, Q_T, \delta)$ *where:*

- Q *is a finite set of states,*
- $Q_T \subseteq Q$ *is a finite set of final states,*
- δ *is a mapping* $Q \times \Sigma \to 2^{Q^*}$ *such that* $\delta(q,a)$ *is a regular language over* Q*.*

An unranked tree t *is accepted by* $\mathcal{H}$ *if there is a run of* $\mathcal{H}$ *on* t *whose root is labeled by a final state.*

There could be different representations of hedge automata depending on how regular expressions over Q are represented. Also, in unranked trees the number of children of a node p in a tree t is not determined by its label, and there is no upper bound on the number of children that a node can have. Therefore automata for unranked trees cannot always be defined by explicitly listing all transition rules in the same style as for ranked tree automata [CDG+08].

Example 2.3.21 *Let* $\mathcal{H} = (Q, \Sigma, Q_T, \Delta)$ *be a hedge automaton where:* $Q = \{q_1, q_2\}$, $\Sigma = \{a, b\}$, $Q_T = \{q_2\}$, *and* $\Delta = \{(a, 1, q_1), (a, q_1, q_2), (b, q_1, q_2, q_2)\}$. *Note that* 1 *is the regular expression of the empty word.*

The tree $t = a(b, a(a, a))$ *is not recognized by this automaton since* b *can't be produced by any transition of* Δ*. But the tree* $t = b(b(a))$ *is recognized by the same automaton by using the first transition to read* a *and get* q_1*, then to get* q_2 *using the second transition* $b(q_1)$*, since* q_2 *can be reached from both* q_1 *and* q_2 *then, reapplying the second transition gives* q_2 *which is a final state.*

Weighted Tree Automata

Weighted finite automata are classical nondeterministic finite automata in which the transitions carry weights. These weights may model, for example, the cost involved when executing a transition, the amount of resources or time needed for this, or the probability or reliability of its successful execution [DKV09]. Then, a tree language becomes a mapping, called a tree series, from the set of trees to a set S of quantities [FV09].

For the definition of weighted automata and their behaviors, matrices and formal power series are used. This makes it possible to use methods of linear algebra over semirings for more succinct, elegant, and convincing proofs [DKV09].

Definition 2.3.22 *Let S be a set and $A = (A, +, , 0, 1)$ be a semiring. A formal power series φ is a mapping $\varphi : S \to A$. Given $s \in S$, we denote $\varphi(s)$ also by (φ, s) and write the series as $\sum_{s \in S} (\varphi, s)s$.*

The support of φ is $supp(\varphi) = \{s \in S | (\varphi, s) \neq 0\}$.

Power series with finite support are called polynomials, and power series with at most one support element are also called singletons [Mal05].

Definition 2.3.23 *Tree series are power series in which the set S is a set of trees.*

Definition 2.3.24 *Let $\phi = (\mathbb{K}, \otimes, \oplus, 0, 1)$ be semiring defined on a set $\mathbb{K}$. Let T_Σ be the set of trees over the alphabet Σ. A tree series is the formal series $\varphi : T \to \mathbb{K}$ where $T \subseteq T_\Sigma$.*

Definition 2.3.25 *A weighted tree automaton is a tuple $\mathcal{A} = (Q, \Sigma, K, \mu, \nu)$ where:*

- *Q is a finite nonempty set of states,*
- *Σ is a ranked alphabet,*
- *K is a semiring,*
- *$\mu = (\mu_k | k \in N)$ is a family of transition mappings $\mu_k : \Sigma^{(k)} \to S^{Q^k} \times Q$, called the weight function,*
- *$\nu : Q \to K$, the root weight vector or the final weight function.*

For every transition $(w, q) \in Q^k \times Q$, the element $\mu_k(\sigma)_{w,q}$ is the weight of (w, q).

The weighted tree automaton $\mathcal{A}$ is deterministic if for every symbol $f \in \Sigma_{(k)}$ and word $w \in Q_k$ there exists at most one state $q \in Q$ such that $\mu_k(f)_{w,q} \neq 0$ [HMV09].

Pushdown Tree Automata

Pushdown automata and context-free grammars are the standard tools to generate and to recognize context-free languages. These languages were first designed to formalize grammatical properties of natural languages, and subsequently they appeared to be well adapted to the formal description of programming languages' syntax, which led to a considerable development of the theory.

Pushdown tree automata extend the well-known string pushdown automata [Har78] by allowing trees instead of strings in both the input and the stack [Sal88]. They have been introduced by Guessarian in [Gue83] and Schimpf et al. in [SG85] who showed that these pushdown tree automata recognize exactly the class of context-free tree languages. Moreover, they showed that pushdown tree automata are equivalent to restricted pushdown tree automata, i.e., to pushdown automata whose pushdown store is linear [Kui01].

The automaton reads its input like a finite top down tree automaton, while scanning the root of the store (the root is the only stack symbol accessible at a given moment) [Gue83].

Definition 2.3.26 *[GS97]*

A top-down pushdown tree automaton is a tuple $A = (Q, \Sigma, \Gamma, P, a_0, z_0)$ *where:*

- Q *is the set of states of* A*.*
- Σ *is the input ranked alphabet.*
- Γ *is the pushdown ranked alphabet.*
- P *is a finite set of productions of the following types:*

  1. $a(f(\xi_i,\ldots,\xi_m),g(\xi_{m+1},\ldots,\xi_{m+n})) \to f(a_1(\xi_1,q_1),\ldots,f(a_m(\xi_m,q_m))$ *with* $a,\ a_1,\ \ldots,\ a_m \in Q,\ f \in \Sigma_m,\ m \geqslant 0,\ g \in \Gamma_n,\ n \geqslant 0,\ q_1,\ldots,q_n \in T_\Gamma(\{\xi_{m+1},\ldots,\xi_{m+n}\})$,
  2. $a(x,g(\xi_1,\ldots,\xi_n)) \to x$, *with* $a \in Q,\ x \in \Sigma_0,\ g \in \Gamma_n,\ n \geqslant 0$),
  3. $a(\xi,g(\xi_1,\ldots,\xi_n)) \to b(\xi,g)$, *with*$(a,b \in Q,\ g \in \Gamma_n,\ n \geqslant 0,\ q \in T_\Gamma(\Xi_n))$.

  First two rules correspond to the reading rules and the second ones to the pushdown recognizers.

- $a_0 \in A$ *is the initial state.*
- $z_0 \in \Gamma_0$ *is the start symbol appearing initially in the pushdown store.*

Example 2.3.27 *Let* $A = (Q, \Sigma, \Gamma, X, P, a_0, z_0)$ *be a pushdown automaton where* $\Sigma = \{f(2), g(1), h(1), x(0)\}$, $Q = a_0$, $\Gamma = \Gamma_0 \cup \Gamma_1$, $\Gamma_0 = \{z_0, z_0'\}$, $\Gamma_1 = \{z_1, z_1'\}$ *and* P *consists of the following productions:*

$a_0(h(\xi), z_0) \to h(q_0(\xi, z_0'))$.

$a_0(f(\xi_1,\xi_2), z_0) \to f(a_0(\xi_1, z_0'), a_0(\xi_2, z_1'(z_1(z_0'))))$,

$a_0(f(\xi_1,\xi_2), z_1'(\xi_3)) \to f(a_0(\xi_1,\xi_3), a_0(\xi_2, z_1'(z_1(\xi_3))))$

$a_0(x, z'_0) \to x,$
$a_0(g(\xi_1), z_1(\xi_2)) \to g(a_0(\xi_1, \xi_2)),$
$a_0(f(\xi_1), z'_1(\xi_2)) \to f(a_0(\xi_1, \xi_2)).$

2.4 Regular Tree Expressions

A Regular Tree Expression (RTE) is a string that allows the specification of a language which has been originally introduced by Kleene [Kle56] for words. It consists basically of symbols of the language alphabet connected with unary (iteration) or binary (union, concatenation) operators. These operators specify, how the resulting tree can be constructed from given symbols. In practice, regular expressions are often used as user interfaces for specifying regular languages [RS97].

Definition 2.4.1 *A* regular tree expression E *over a ranked alphabet* Σ *is inductively defined by:*

- $\mathrm{E} = 0,$
- $\mathrm{E} \in \Sigma_0,$
- $\mathrm{E} = f(\mathrm{E}_1, \cdots, \mathrm{E}_n),$
- $\mathrm{E} = \mathrm{E}_1 + \mathrm{E}_2,$
- $\mathrm{E} = \mathrm{E}_1 \cdot_c \mathrm{E}_2,$
- $\mathrm{E} = \mathrm{E}_1{}^{*_c}.$

where $c \in \Sigma_0$, $n \in \mathbb{N}$, $f \in \Sigma_n$ *and* $\mathrm{E}_1, \mathrm{E}_2, \ldots, \mathrm{E}_n$ *are any* n *regular expressions over* Σ. *We call* $\mathrm{E} = f(\mathrm{E}_1, \cdots, \mathrm{E}_n)$ *the arity operation.*

Definition 2.4.2 *The regular tree language* $[\![\mathrm{E}]\!]$ *denoted by the regular tree expression* E *is inductively defined by:*

- $[\![0]\!] = \emptyset,$
- $[\![c]\!] = \{c\},$
- $[\![f(\mathrm{E}_1, \mathrm{E}_2, \cdots, \mathrm{E}_n)]\!] = f([\![\mathrm{E}_1]\!], \ldots, [\![\mathrm{E}_n]\!]),$
- $[\![\mathrm{E}_1 + \mathrm{E}_2]\!] = [\![\mathrm{E}_1]\!] \cup [\![\mathrm{E}_2]\!],$
- $[\![\mathrm{E}_1 \cdot_c \mathrm{E}_2]\!] = [\![\mathrm{E}_1]\!] \cdot_c [\![\mathrm{E}_2]\!],$

- $[\![\mathrm{E}_1{}^{*_c}]\!] = [\![\mathrm{E}_1]\!]^{*_c}$.

where $n \in \mathbb{N}$, $\mathrm{E}_1, \mathrm{E}_2, \ldots, \mathrm{E}_n$ *are any* n *regular expressions,* $f \in \Sigma_n$ *and* $c \in \Sigma_0$.

Property 2.4.3 *A tree language is accepted by some tree automaton if and only if it can be denoted by a regular tree expression [CDG+08, KM11].*

Definition 2.4.4 *The size of the regular tree expression* E*, denoted by* $|E|$ *is defined inductively by:*

- $|c| = 1$ *for* $c \in \Sigma_0$,
- $|f(E_1, \ldots, E_m)| = \sum_{i=1}^{m} |E_i| + 1$ *for* $f \in \Sigma_m$,
- $|E + F| = |E \cdot_c F| = |E| + |F| + 1$,
- $|E^{*_c}| = |E| + 1$.

Every regular tree expression E can be seen as a tree over the ranked alphabet $\Sigma \cup \{+, \cdot_c, *_c \mid c \in \Sigma_0\}$ where + and $\cdot_c$ can be seen as symbols of rank 2 and $*_c$ has rank 1.

Example 2.4.5 $E = (f(a,b) + g(c) \cdot_c d)^{*_d}$ *is an example of a regular tree expression on the ranked alphabet* $\Sigma = \{a(0), b(0), c(0), d(0), g(1), f(2)\}$.

2.5 Regular Tree Grammars

For some applications like model checking or language synthesis, it is more convenient to have a generative representation than an accepting one. In regular tree languages, the generative representation is the regular tree grammar. It is considered as a generalization of string grammars as well [Hög07].

Definition 2.5.1 *A regular tree grammar* G *is a quadruple* (N, Σ, P, S)*, where*

- N *is ranked alphabet of nonterminals,*
- Σ *is an alphabet of terminals such that* $N \cap \Sigma = \emptyset$,
- P *is a finite set of productions of the form* $X \to t$*, where* $X \in N$ *and* $t \in \Sigma \cup N$,
- $S \subseteq N$ *is the set of starting nonterminals.*

Example 2.5.2 *Let* $N = \{X, A\}$, $\Sigma = \{a, b, c\}$, $S = \{X\}$ *and* $P = \{X \to b(a(c,c)), X \to b(X)\}$. $G = (N, \Sigma, P, A)$ *is a regular tree grammar whose language is* $\mathcal{L}_G$.

$\mathcal{L}_G = \{b(a(c,c)), b(b(a(c,c))), b(b(b(a(c,c)))), ...\}$.

Definition 2.5.3 *Let* $G = (N, \Sigma, P, S)$ *be a regular tree grammar, and let* s *and* s' *be trees in* T_Σ. *A derivation step in* G *from* s *to* s' *denoted by* $s \Rightarrow_G s'$ *can be obtained by replacing one occurrence of a nonterminal in* s *according to a production in* P.

Example 2.5.4 *Using the grammar of the last example,* $X \Rightarrow b(X) \Rightarrow b(b(a(c,c)))$.

Definition 2.5.5 *We denote the transitive closure of* G *by* $\Rightarrow_G^*$, *and the language* $\mathcal{L}(G)$ *generated by* G *by* $\mathcal{L}(G) = \{t \in T_\Sigma | X' \Rightarrow_G^* t$ *for some* $X' \in X\}$.

Definition 2.5.6 *We say that a regular tree grammar is in normal form iff each of its productions is of the form* $X \to f[X_1, \ldots, X_k]$, *where* f *is a terminal of rank* k, *and* $X, X_1, \ldots, X_k$ *are nonterminals.*

2.6 Applications of Regular Tree Languages

There exist mainly three important problems for regular tree languages: tree acceptance, tree pattern matching and tree parsing.

Tree Acceptance

Tree acceptance is the basic problem of regular tree languages. It is generally solved by first constructing a tree automaton based on a regular tree expression or a regular tree grammar then this automaton is run on an input tree. If the automaton accepts the tree, then the tree can be generated by the regular expression or the regular grammar.

Definition 2.6.1 *Given a regular tree grammar* G *and an input tree* $t \in T_\Sigma$. *Tree acceptance is to determine whether* t *can by generated by* G.

Tree Pattern Matching

Tree pattern matching is the process that consists of finding all occurrences of a given tree pattern in a subject tree. It is an important operation on which a number of tasks in computer science are based on, i.e. automatic theorem proving, term-rewriting, instruction selection and non-procedural programming languages. In addition, tree pattern matching has direct applications in computational biology [Flo12].

There exist two types of tree pattern matching: subtree and tree template matching. While subtrees consist of only specific, fixed trees, tree templates may have variable leaves, that is leaves with a symbol which can be substituted by any other subtree of the tree language. This symbol is referred to as υ.

Definition 2.6.2 *For a tree $t \in T_\Sigma$ and a pattern $p \in T_{\Sigma\cup\{\upsilon\}}$, we say that a subtree in t matches the pattern p in a node n means that $p = t/n$, where t/n is the subtree of t rooted by n.*

Tree parsing

The tree parsing is a variant of the tree acceptance problem except that it does not only decide whether a tree can be produced by a grammar, but also how this production is done. That is to say, which production rule needs to be applied for each node in the input tree. Another variation of this problem is to determine the optimal parse with respect to predefined cost function [Cle08].

Definition 2.6.3 *Given a regular tree grammar G and an input tree $t \in T_\Sigma$. Tree parsing consists on determining all parses of t that can be generated by G.*

2.7 Discussion and Conclusion

In this chapter we have presented the basic definitions and results in the field of tree languages, especially tree automata. For more details and proofs we refer to [Tha73, Eng15, GS15, GS97, CDG+08]. As it has been shown, a lot of concepts for the theory of automata over finite words have been adapted to ranked trees, in particular automata construction and their properties, minimization of automata, ... Also from an algorithmic point of view there are a lot of similarities to word automata, where in most cases the cost for algorithms on tree automata is exponentially higher than for word automata.

There are various active research areas on different new aspects of tree automata. A particular tree automata domain motivated from the XML world is the extension of automata to deal with data values from infinite domains, that is unranked or hedge tree automata. Another type of tree automata we omitted to mention is the tree transducers. The concept of a tree transducer is based on a function that maps one tree to another in a possibly nondeterministic manner. The most basic types of tree transducers were introduced soon after the first papers on tree automata and

tree grammars. In particular, Rounds and Thatcher invented the top-down and the bottom-up tree transducer [Rou70, Tha70].

There exist other formalisms than tree automata used to represent regular tree languages like regular tree grammars, regular tree expressions and regular tree systems. While the regular tree grammars and regular tree expressions are a generalization of word grammars and word regular expressions, tree systems are a more generalized version of tree grammars since multiple starting symbol may exist, nonterminals might have a rank instead of only constant, ... We have not treated these two formalisms in this chapter as they are not needed later.

There is a significant collection of application areas where the three problems in the regular tree language domain play a role [Cle08]:

- Code generation in compilers: instruction selection or optimization,
- term rewriting and unification,
- genetics: DNA/RNA pattern matching,
- XML document processing,
- type inference,
- protocol verification: cryptography and network protocols.

In the two next chapters we treat a very important aspect of regular tree languages, which is the equivalence of tree automata and regular tree expressions languages. This problem is known as the Kleene theorem, it has been initiated in the word theory and later proved for trees. This theorem has been useful for converting regular tree expressions to tree automata and back.

Chapter 3

From Regular Tree Expressions to Tree Automata and Back: State of the Art

In the case of words, Kleene [Kle56] has shown that the family of regular languages and the family of deterministic finite automata languages are exactly the same, i.e. regular expressions are equivalent to finite automata in terms of the languages they define [GS97]. This result has been successfully verified for tree languages [CDG+08].

It is well known that regular expressions are well suited for human users and therefore are often used as interfaces to specify patterns or to define regular languages. For example, in UNIX , regular expressions can be found in many tools like, ed, emacs, grep, sed, vi, etc. On the other hand, automata [RS59] immediately translate to efficient data structures, and are convenient for programming tasks [GH15]. This naturally shows how important is the conversions between these two different formalisms.

In the current chapter we will describe both approaches for the transformation from a regular tree expression to an equivalent tree automaton and back. We give the main idea of these approaches in the word case and then we detail their generalizations to trees. Figure 3.1 shows the classification of the algorithms treated in this chapter according to whether they convert tree automata to regular tree expression or the contrary.

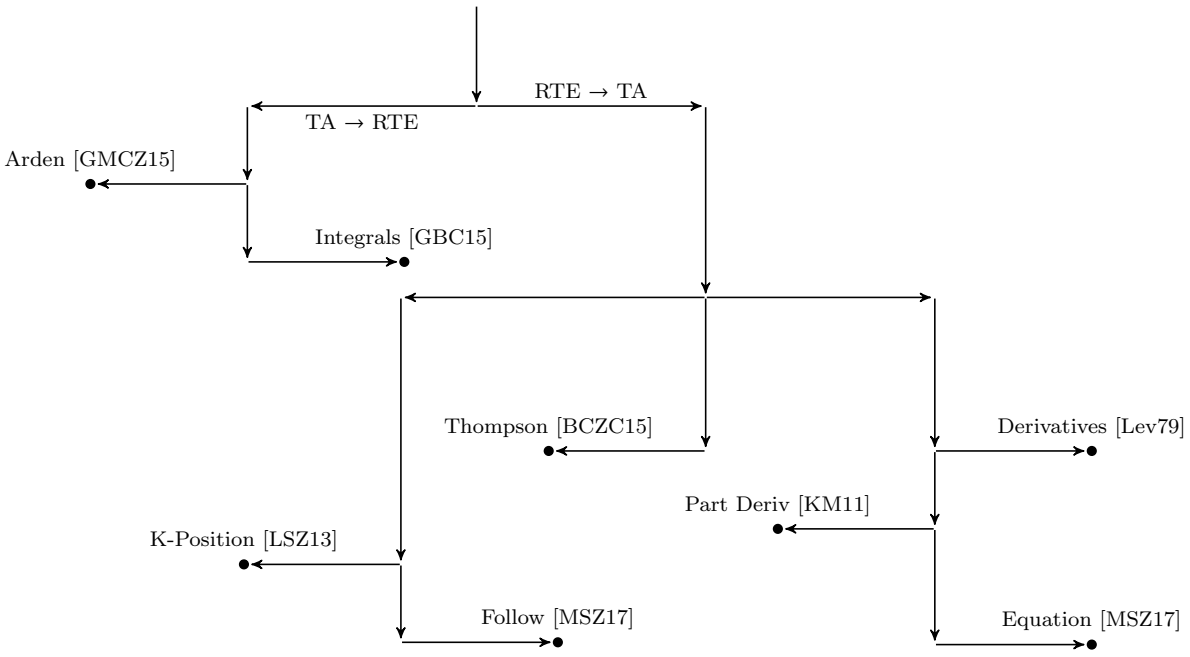


Figure 3.1: Regular tree expression to tree automata and back algorithms.

3.1 Part One: From Regular Tree Expressions to Tree Automata

For words, the conversion of regular expressions into finite automata has been intensively studied [GH15]. Basically the algorithms can be classified according to whether the output automaton is nondeterministic with ε−transitions (ε−NFA), nondeterministic without ε−transitions (NFA), or even a deterministic one (DFA). There exist mainly three major constructions:

1. Thompson's construction [Tho68] and optimized versions, such as the follow automaton [IY03],
2. the (partial) derivative automaton [Ant96, Brz64, CZ01, CZ02], and
3. the position automaton, or Glushkov automaton [Glu61, MY60].

Other word automata constructions from regular expressions can be found in [ACT10, BK93, CP92, HSW01, GLRÁ11, Wat95, OF61], which are more or less an optimization of the previous constructions. For further constructions in words we refer to [ST09].

The first approach, due to Thompson [Tho68], is to transform a regular expression into a ε-NFA. This approach is simple and intuitive, it consists of building an automaton inductively on the regular expression. The main disadvantage of this approach is that it may generate many ε-transitions. Thus, the resulting NFA can be very large and its transformation into a DFA needs additional time and space [RS97]. On the other hand, this approach is well known for being used in the string pattern matching tool **grep** in UNIX-like OS [Goe95, AA04] because of its reliability and simple implementation.

The second approach is to transform a regular expression directly into a DFA using derivatives [Brz64]. This approach can be replaced by two separate steps: the conversion of the regular expressions into an NFA using any approach and then determinizing the resulting automaton [RS97].

The third approach transforms a regular expression into an NFA without ε-transitions. This approach is due to Berry and Sethi [BS86], whose algorithm is based on Brzozowski's theory of derivatives [Brz64] and McNaughton and Yamada's marked expression algorithm [MY60]. Berry and Sethi's algorithm has been further improved by Brüggemann-Klein [BK93] and Chang and Paige [CP92].

By analogy to words, multiple algorithms have been proposed for trees. Laugerotte et al. [LSZ13] gave an algorithm to compute the position tree automaton. Tree

derivatives were introduced by Levine in [Lev81, Lev82] and extend the concept of Brzozowski's string derivatives [Brz64]. The work of Kuske and Meinecke [KM11] consists of the definition of partial derivatives for regular tree expressions and then building a nondeterministic finite tree automaton recognizing the language denoted by such an expression. They have adapted and modified the approach of Champarnaud and Ziadi [CZ01, CZ02] in the word case.

Hereafter, we give briefly the different tree automata constructions from regular tree expressions with examples. We mention that the reader may skip this chapter of the state of the art as it is not needed to understand our contributions.

Definition 3.1.1 *The linearized regular expression $\overline{E}$ is a regular expression obtained from E by marking differently all symbols of $\Sigma_{\geqslant 1}$. The marked symbols form together with the constants in Σ_0 a ranked alphabet $Pos_E(E)$ called alphabet of positions.*

Definition 3.1.2 *The mapping h is defined from $Pos_E(E)$ to Σ with $h(Pos_E(E)m) \subset \Sigma_m$ for every $m \in N$. It associates with a marked symbol $f_j \in Pos_E(E) \geqslant 1$ the symbol $f \geqslant 1$ and for a symbol $c \in \Sigma_0$ the symbol $h(c) = c$. This mapping is extended to regular expressions.*

Example 3.1.3 *Let $\Sigma = \{a, b, c, f(1), h(1), g(2)\}$. Let the regular expression $E = (f(a)^{*a}_{.a}b + h(b))^{*b} + g(c,a)^{*c}_{.c}(f(a)^{*a}_{.a}b + h(b))^{*b}$ and its linearized form $\overline{E} = (f_1(a)^{*a}_{.a}b + h_2(b))^{*b} + g_3(c,a)^{*c}_{.c}(f_4(a)^{*a}_{.a}b + h_5(b))^{*b}$.*

3.1.1 K-Position Tree Automata

This construction is an extension of the well-known position automaton [Glu61] for word regular expressions where K represents the fact that any k-ary symbol is no longer a state of the automaton, but is exploded into k states. Its computation is based on the definition of particular position functions, defined hereafter [LSZ13, MSZ14b].

In the following, we denote for any two trees t and s the fact that s is a subtree of t by $s \preccurlyeq t$. For a tree $t = f(t_1, \ldots, t_n)$, we denote by $root(t)$ its root, k-child(t) the k$^{\text{th}}$ child of f in t and by $Leaves(t)$ the set of leaves of t, that is $\{s \in \Sigma_0 | s \preccurlyeq t\}$.

Let E be a linear regular expression, m and k two integers such that $1 \leqslant k \leqslant m$, and $f \in \Sigma_m$. We define the following sets:

- $First(E) \subseteq \Sigma$, such that $First(E) = \{root(t) | t \in [\![E]\!]\}$.
- $Follow(E, f, k) \subseteq \Sigma$, such that:

  $Follow(E, f, k) = \{g \in \Sigma | \exists t \in [\![E]\!], \exists s \preccurlyeq t, root(s) = f, k - child(s) = g\}$.

- $Last(E) \subseteq \Sigma_0$, such that $Last(E) = \{ \bigcup_{t \in [\![E]\!]} Leaves(t) \}$.

Example 3.1.4 *We consider the last example 3.1.3. The language denoted by* $\overline{E}$ *is* $[\![\overline{E}]\!] = \{b, f_1(b), f_1(f_1(b)),\ f_1(h_2(b)),\ h_2(b),\ h_2(f_1(b)),\ h_2(h_2(b)),\ \ldots,\ g_3(b,a),\ g_3(g_3(b,a),a),\ g_3(f_4(b),a),\ g_3(h_5(b),a),\ f_4(f_4(b)),\ f_4(h_5(b),$

$h_5(f_4(b)),\ h_5(h_5(b)),\ \ldots\}$.

Consequently,

- $First(\overline{E}) = \{b, f_1, h_2, g_3, f_4, h_5\}$,
- $Follow(\overline{E}, f_1, 1) = \{b, f_1, h_2\}$, $Follow(\overline{E}, h_2, 1) = \{b, f_1, h_2\}$,

  $Follow(\overline{E}, g_3, 1) = \{b, g_3, f_4, h_5\}$, $Follow(\overline{E}, g_3, 2) = \{a\}$, $Follow(\overline{E}, f_4, 1) = \{b, f_4, h_5\}$, $Follow(\overline{E}, h_5, 1) = \{b, f_4, h_5\}$.

The functions $First$ and $Follow$ can be computed inductively, and are sufficient to construct the K-position automaton [MSZ14b, MSZ17].

Definition 3.1.5 *Let* E *be a linear regular tree expression. The* K*-position automaton* $\mathcal{P}_E = (Q, \Sigma, Q_T, \Delta)$ *is defined as follows:*

- $Q = \{f^k | f \in \Sigma_m \wedge 1 \leq k \leq m\} \cup \{\varepsilon^1\}$ *with* ε^1 *a new symbol not in* Σ,
- $Q_T = \{\varepsilon^1\}$,
- $\Delta = \{(f^k, g, g^1, \ldots, g^n) | f \in \Sigma_m \wedge k \leq m \wedge g \in \Sigma_n \wedge g \in Follow(E, f, k)\}$

  $\cup \{(\varepsilon^1, f, f^1, \ldots, f^m) | f \in \Sigma_m \wedge f \in First(E)\}$

  $\cup \{(\varepsilon^1, c) | c \in \Sigma_0 \wedge c \in First(E)\}$

  $\cup \{(f^k, c) | f \in \Sigma_m \wedge k \leq m \wedge c \in Follow(E, f, k)\}$

Example 3.1.6 *Let* $E = (f(a)^{*a}_{.a} b + h(b))^{*b} + g(c,a)^{*c}_{.c} (f(a)^{*a}_{.a} b + h(b))^{*b}$ *be the regular expression of Example 3.1.3. The k−Position Automaton* $\mathcal{P}_{\overline{E}}$ *associated with* $\overline{E}$ *is given in Figure 3.2. The set of states is* $Q = \{\varepsilon^1, f_1^1, h_2^1, g_3^1, g_3^2, f_1^4, h_5^1\}$. *The set of final states is* $Q_T = \{\varepsilon^1\}$. *The set of transition rules* Δ *is :*

$f_1(f_1^1) \to f_1^1$	$f_1(f_1^1) \to \varepsilon^1$	$f_1(f_1^1) \to h_2^1$
$h_2(h_2^1) \to \varepsilon^1$	$h_2(h_2^1) \to f_1^1$	$h_2(h_2^1) \to h_2^1$
$g_3(g_3^1, g_3^2) \to g_3^1$	$g_3(g_3^1, g_3^2) \to \varepsilon^1$	$f4(f_4^1) \to \varepsilon^1$
$f_4(f_4^1) \to g_3^1$	$f_4(f_4^1) \to f_4 1$	$f_4(f_4^1) \to h_5^1$
$h_5(h_5^1) \to \varepsilon^1$	$h_5(h_5^1) \to g_3^1$	$h5(h_5^1) \to f_4^1$
$h_5(h_5^1) \to h_5^1$	$a \to g_3^2$	$b \to \varepsilon^1$
$b \to f_1^1$	$b \to h_2^1$	$b \to g_3^1$
$b \to f_4^1$	$b \to h_5^1$	

The number of states is $|Q| = 7$ *and the number of transition rules is* $|\Delta| = 23$.

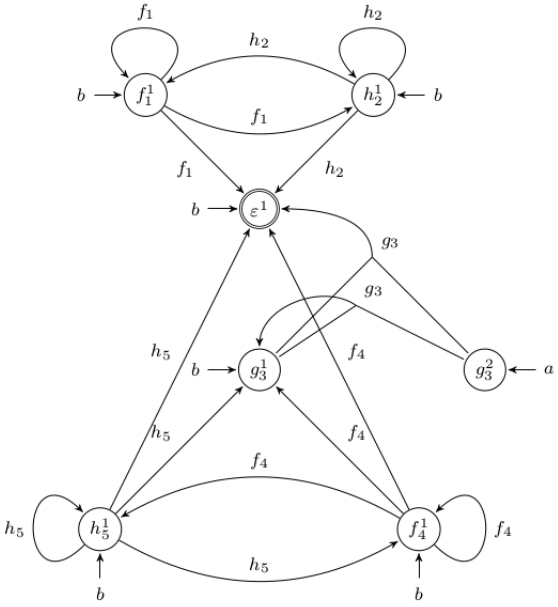


Figure 3.2: K−Position automaton of Example 3.1.6 [MSZ17]

3.1.2 Follow Tree Automata

The Follow tree automaton is a generalization of the word Follow automaton introduced by Ilie and Yu [IY03]. Notice that in this automaton, states are no longer positions, but sets of positions.

Definition 3.1.7 *Let* E *be a linear regular tree expression. The Follow Automaton of* E *is the tree automaton* $\mathcal{F}_E = (Q, \Sigma, Q_T, \Delta)$ *defined as follows:*

- $Q = \{First(E)\} \cup \{ \bigcup_{f \in \Sigma_{E_m}} \{Follow(E,f,k) \mid 1 \leq k \leq m\}\}$,
- $Q_T = First(E)$,
- $\Delta = \{(Follow(E,g,l),\ f, Follow(E,f,1), \ldots,\ Follow(E,f,m)\ | f \in \Sigma_{E_m}$
  $\wedge f \in Follow(E,g,l) \wedge g \in \Sigma_n \wedge l \leq n\} \cup (I,c) | c \in I \wedge c \in \Sigma_0$.

In the work of Mignot et al. [MSZ14b, MSZ17], it has been shown that the language denoted by a regular expression E is recognized by $\mathcal{F}_E$ and that this automaton is a quotient of the K-position automaton.

Example 3.1.8 *The Follow Automaton* $\mathcal{F}_E$ *associated with* $E = (f(a)^{*a}_{.a} b + h(b))^{*b} + g(c,a)^{*c}_{.c} (f(a)^{*a}_{.a} b + h(b))^{*b}$ *of Example 3.1.3 is given in Figure 3.3.*

The set of states is $Q = \{\{a\}, \{b, f_1, h_2\}, \{b, f_1, h_2, g_3, f_4, h_5\}, \{b, g_3, f_4, h_5\}, \{b, f_4, h_5\}\}$ *and* $Q_T = \{\{b, f_1, h_2, g_3, f_4, h_5\}\}$.

The set of transition rules Δ *is:*

$f(\{b, f_1, h_2\}) \to \{b, f_1, h_2\}$ $\quad$ $f(\{b, f_4, h_5\}) \to \{b, f_1, h_2, g_3, f_4, h_5\}$

$f(\{b, f_4, h_5\}) \to \{b, g_3, f_4, h_5\}$ $\quad$ $f(\{b, f_4, h_5\}) \to \{b, f_4, h_5\}$

$f(\{b, f_1, h_2\}) \to \{b, f_1, h_2, g_3, f_4, h_5\}$

$h(\{b, f_4, h_5\}) \to \{b, f_1, h_2, g_3, f_4, h_5\}$ $\quad$ $h(\{b, f_4, h_5\}) \to \{b, f_4, h_5\}$

$h(\{b, f_1, h_2\}) \to \{b, f_1, h_2, g_3, f_4, h_5\}$ $\quad$ $h(\{b, f_1, h_2\}) \to \{b, f_1, h_2\}$

$h(\{b, f_4, h_5\}) \to \{b, g_3, f_4, h_5\}$

$g(\{b, g_3, f_4, h_5\}, \{a\}) \to \{b, g_3, f_4, h_5\}$

$g(\{b, g_3, f_4, h_5\}, \{a\}) \to \{b, f_1, h_2, g_3, f_4, h_5\}$

$b \to \{b, g_3, f_4, h_5\}$ $\quad$ $b \to \{b, f_1, h_2, g_3, f_4, h_5\}$

$b \to \{b, f_1, h_2\}$ $\quad$ $b \to \{b, f_4, h_5\}$

$a \to \{a\}$

The number of states is $|Q| = 5$ *and the number of transition rules is* $|\Delta| = 17$.

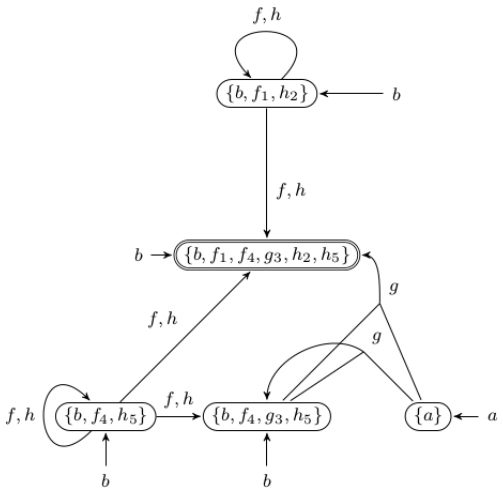


Figure 3.3: Follow automaton of Example 3.1.6 [MSZ17]

3.1.3 Tree Derivatives

Levine [Lev79, Lev81, Lev82] is the first who introduced the notion of tree derivatives which extends the concept of Brzozowski's string derivatives [Brz64]. This extension has been used for minimizing and characterizing tree automata. Tree derivatives are used as a basis of inference of tree automata from finite samples of trees (rather than regular tree expressions) by comparing tree derivative sets and inferring a tree automaton based on the amount of overlap between the derivative sets.

Definition 3.1.9 *The derivative of a tree* t *with respect to* u*, denoted* $d_u(t)$*, is defined as follows:*

$$d_u(t) = \begin{cases} t' | t' = t(b_1 \leftarrow \$), \ldots, (b_m \leftarrow \$) & \text{if } u \text{ is a subtree of } t \text{ at } b_1, \ldots, b_m \\ \emptyset & \text{otherwise} \end{cases}$$

Hence $d_u(t)$ is the tree resulting from the replacement of each occurrence of the subtree u, of t with \$, i.e. the pruning of the subtrees u.

Example 3.1.10 *Let $t = a(b(c,d),d(b(c,d),c))$, $u = b(c,d)$ and $v = d(b,c)$ be trees. then*

$d_u(t) = a(\$, d(\$, c))$.

$d_v(t) = \emptyset$.

The derivative of a tree is extended to a set of trees as follows.

Definition 3.1.11 *Let $S = \{t_1, \ldots, t_n\}$ be a set of trees over an alphabet Σ. Then the derivative of S with respect to u, denoted by $D_u(S)$, is defined as:*

$$D_u(S) = \bigcup_{i=1}^{n} d_u(t_i)$$

This definition allows the replacement of all subtrees u by \$. The following one defines the replacement of a given number of the same subtree with \$.

Definition 3.1.12 *Let $S = \{t_1, \ldots, t_n\}$ and $\{t\}$ be sets of trees over $A \cup \{\$\}$. Let b_i and c_j^i describe the occurrences of t in trees of S as follows: b_i the number of occurrences of t in t_i and c_j^i is the j^{th} occurrence of t in t_i.*

Thus, c_j^i is defined for $c_1^i, c_2^i, \ldots, c_{b_i}^i$ where c_j^i corresponds to a node in the tree domain of t_i.

The m^{th} derivative of S with respect to t is defined recursively as:

- $D_t^0(S) = S$,
- $D_t^1 = \bigcup_{i=1}^{n} \begin{cases} \bigcup_{j=1}^{b_i} t_i(c_j^i \leftarrow \$) & \text{if } b_i \geq 1 \\ \emptyset & \text{otherwise} \end{cases}$
- $D_t^m(S) = D_t^1(D_t^{m-1}(S))$ *if* $m > 1$.

Example 3.1.13 *Let $S = \{a(b(c,d),d(b(c,d),c)), a(b(c,d),c)\}$ and $t = b(c,d)$.*

$D_t^1(S) = \{a(\$, d(b(c,d),c)), a(b(c,d), d(\$,c)), a(\$,c)\}$

$D_t^2(S) = D_t^1(D_t^1(S)) = \{a(\$, d(\$,c))\}$

The derivative of a set of trees serves to exhibit various characteristics of trees belonging to the same equivalence class. However, the derivative is not sufficient to completely characterize related trees. Powers of derivatives must be applied in order to get the properties of trees in the same equivalence class [Lev79].

Levine used tree derivatives in a state minimization algorithm for tree automata and then proved that a tree automaton M with at most n states can be characterized by the set of trees it accepts of depth at most 2^n.

3.1.4 Equation Tree Automata

In [KM11], Kuske and Meinecke extend the notion of word partial derivatives [Ant96] to tree partial derivatives in order to compute from a regular tree expression E, a tree automaton recognizing $[\![E]\!]$. Due to the notion of ranked alphabet, partial derivatives are no longer sets of expressions, but sets of tuples of expressions.

Before defining the partial derivatives of a regular tree expression, we give hereafter some definitions. Let $\mathcal{N} = (E_1, \ldots, E_n)$ be a tuple of regular expressions, F and G be some regular expressions and $c \in \Sigma_0$.

Definition 3.1.14 *The tuple $\mathcal{N}._c F$ is obtained by concatenating F to $E_1, \ldots, E_n$. That is $(E_1._c F, \ldots, E_n._c F)$.*

Definition 3.1.15 *For a set $\mathcal{S}$ of tuples of regular expressions, $\mathcal{S}._c F$ is the set $\mathcal{S}._c F = \{\mathcal{N}._c F | \mathcal{N} \in \mathcal{S}\}$.*

We put $SET(\mathcal{N}) = E_1, \ldots, E_m$ and $SET(\mathcal{S}) = \bigcup_{\mathcal{N} \in \mathcal{S}} SET(\mathcal{N})$.

Definition 3.1.16 *Let f be a symbol in $\Sigma_{>0}$. The set $f^{-1}(E)$ of tuples of regular expressions is defined as follows:*

- $f^{-1}(0) = \emptyset$,
- $f^{-1}(F + G) = f^{-1}(F) \cup f^{-1}(G)$,
- $f^{-1}(F^{*c}) = f^{-1}(F)._c F^{*c}$,
- $f^{-1}(g(E_1, \ldots, E_n)) = (E_1, \ldots, E_n)$ *if* $f = g$ *and* $\emptyset$ *otherwise,*
- $f^{-1}(F._c G) = f^{-1}(F)._c G$ *if* $c \notin [\![F]\!]$ *and* $f^{-1}(F)._c G \cup f^{-1}(G)$ *otherwise.*

The function f^{-1} is extended to any set S of regular expressions by $f^{-1}(S) = \bigcup_{E \in S} f^{-1}(E)$.

Definition 3.1.17 *The partial derivative of a regular tree expression E w.r.t. a word $w \in \Sigma^*_{\geqslant 1}$, denoted by $\partial_w(E)$, is the set of regular expressions inductively defined by:*

- $\{E\}$ *if* $E = \varepsilon$,
- $SET(f^{-1}(\partial_u(E)))$ *if* $w = uf, f \in \Sigma_{\geqslant 1}, u \in \Sigma^*_{\geqslant 1}, f^{-1}(\partial_u(E)) \neq \emptyset$,
- $\{0\}$ *if* $w = uf, f \in \Sigma_{\geqslant 1}, u \in \Sigma^*_{\geqslant 1}, f^{-1}(\partial_u(E)) = \emptyset$.

Definition 3.1.18 *Let E be a linear regular tree expression. The Equation Automaton of E is the tree automaton $\mathcal{E}_E = (Q, \Sigma, Q_T, \Delta)$ defined by:*

- $Q = \{\partial_w(E) | w \in \Sigma^*_{\geqslant 1}\}$,
- $Q_T = \{E\}$,
- $\Delta = \{(F, f, G_1, \ldots, G_m) | F \in Q, f \in \Sigma_m, m \geqslant 1, (G_1, \ldots, G_m) \in f^{-1}(F)\} \cup \{(F, c) | F \in Q \wedge c \in (\llbracket F \rrbracket \cap \Sigma_0)\}$.

Example 3.1.19 *The equation Automaton $\mathcal{E}_E$ associated with the RTE of Example 3.1.3, $E = (f(a)^{*a}_{.a}b + h(b))^{*b} + g(c,a)^{*c}_{.c}(f(a)^{*a}_{.a}b + h(b))^{*b}$ is given in Figure 3.4.*

Let $E = \underbrace{(f(a)^{*a}_{.a}b + h(b))^{*b}}_{F} + \underbrace{g(c,a)^{*c}}_{G}{}_{.c}\, \underbrace{(f(a)^{*a}_{.a}b + h(b))^{*b}}_{F}$.

$\partial_h(E) = \{b_{.b}F\}$, $\partial_f h(E) = \{b_{.b}F\}$, $\partial_g h(E) = \{b_{.b}F\}$.

$\partial_f(E) = \{((a_{.a}f(a)^a)_{.a}b)_{.b}F\}$, $\partial_g(E) = \{(a_{.c}G)_{.c}F, (c_{.c}G)_{.c}F\}$,

$\partial_{hh}(E) = \{((a_{.a}f(a)^a)_{.a}b)_{.b}F\}$, $\partial_{ff}(E) = \{((a_{.a}f(a)^a)_{.a}b)_{.b}F\}$,

$\partial_{hf}(E) = \{((a_{.a}f(a)^a)_{.a}b)_{.b}F\}$, $\partial_{gf}(E) = \{((a_{.a}f(a)^a)_{.a}b)_{.b}F\}$,

$\partial_g g(E) = \{(a_{.c}G)_{.c}F, (c_{.c}G)_{.c}F\}$.

The set of states $Q = \{q_0 = E, q_1 = ((a_{.a}f(a)_a)_{.a}b)_{.b}F, q_2 = b_{.b}F, q_3 = (c_{.c}G)_{.c}F, q_4 = (a_{.c}G)_{.c}F\}$.

The set of final states is $Q_T = \{q_0\}$.

The set of transition rules is:

$f(q_1) \to q_0$ $\quad$ $f(q_1) \to q_1$
$f(q_1) \to q_2$ $\quad$ $f(q_1) \to q_4$
$h(q_2) \to q_0$ $\quad$ $h(q_2) \to q_1$
$h(q_2) \to q_2$ $\quad$ $h(q_2) \to q_4$
$g(q_3, q_4) \to q_0$ $\quad$ $g(q_3, q_4) \to q_4$
$b \to q_0$ $\quad$ $b \to q_1$
$b \to q_3$ $\quad$ $b \to q_2$
$a \to q_4$

The number of states is $|Q| = 5$ *and the number of transition rules is* $|\Delta| = 15$.

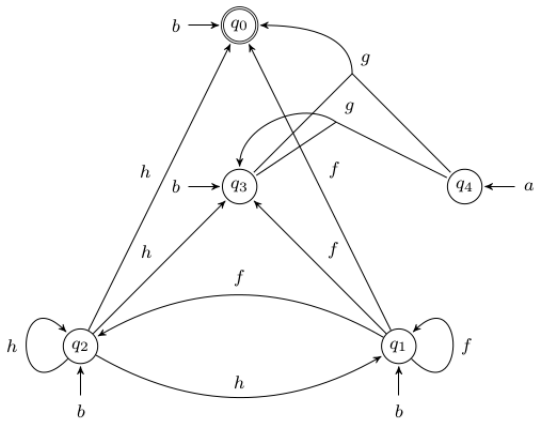


Figure 3.4: Equation automaton of Example 3.1.3 [MSZ17]

3.1.5 K-C-Continuation Tree Automata

In[KM11], Kuske and Meinecke show how to efficiently compute the equation tree automaton of a regular expression via an extension of Champarnaud and Ziadi's C-Continuation [CZ01, CZ02, KOZ08]. In [MSZ14a], it has been shown how to inductively compute them and how to efficiently compute the k-C-Continuation tree automaton associated with a regular expression [MSZ14b, MSZ17].

Definition 3.1.20 *Let* $E \neq 0$ *be a linear regular tree expression. Let* k *and* m *be two integers such that* $1 \leqslant k \leqslant m$. *Let* $f \in \{\Sigma_E \cap \Sigma_m\}$. *The k-C-continuation* $C_{f^k}(E)$ *of* f *in* E *is the regular expression defined by:*

- $C_{f^k}(g(E_1, \ldots, E_m)) = E_k$ *if* $f = g$ *and* $C_{f^k}(E_j)$ *if* $f \in \Sigma_{E_j}$,
- $C_{f^k}(E_1 + E_2) = C_{f^k}(E_1)$ *if* $f \in \Sigma_{E_1}$ *and* $C_{f^k}(E_2)$ *if* $f \in \Sigma_{E_2}$,

- $C_{f^k}(E_{1.c}E_2) = C_{f^k}(E_1)_{.c}G$ *if* $f \in \Sigma_{E_1}$, $C_{f^k}(E_2)$ *if* $f \in \Sigma_{E_2} \wedge c \in Last(E_1)$ *and* 0 *otherwise,*
- $C_{f^k}(F^c) = C_{f^k}(F)_{.c}F^c$.

By convention, $C_{\varepsilon^1}(E) = E$.

Definition 3.1.21 *Let* $E \neq 0$ *be a linear regular tree expression. The automaton* $C_E = (Q_C, \Sigma_E, C_{\varepsilon^1}(E), \Delta_C)$ *is defined by:*

- $Q_C = (f^k, C_{f^k}(E))|f \in \Sigma_m, 1 \leqslant k \leqslant m \cup (\varepsilon^1, C_{\varepsilon^1}(E))$,,
- $\Delta_C = \{((x, C_x(E)), g, ((g_1, C_{g^1}(E)), \ldots, (g_m, C_{g^m}(E))))|g \in \Sigma_{Em}$
  $m \geqslant 1, (C_{g^1}(E), \ldots, C_{g^m}(E)) \in g^{-1}(C_x(E))\} \cup ((x, C_x(E)), c)|, c \in [\![C_x(E)]\!] \cap \Sigma_0$
  .

The K-C-Continuation tree automaton C_E associated with E is obtained by relabeling the transitions of $C_{\overline{E}}$ using the mapping h (Definition 3.1.2).

Example 3.1.22 *Computation of the k-C-Continuations of E of Example 3.1.3:*

$C_{f_1^1}(\overline{E}) = ((a_{.a}f_1(a)^{*a})_{.a}b)_{.b}F_1$ $\quad h(C_{f_1^1}(\overline{E})) = ((a_{.a}f(a)^{*a})_{.a}b)_{.b}F$,
$C_{h_2^1}(\overline{E}) = b_{.b}F_1$ $\quad h(C_{h_2^1}(\overline{E})) = b_{.b}F$
$C_{g_3^1}(\overline{E}) = (c_{.c}g_3(c,a)^{*c})_{.c}F_3$ $\quad h(C_{g_3^1}(\overline{E})) = (c_{.c}g(c,a)^{*c})_{.c}F$
$C_{g_3^2}(\overline{E}) = (a_{.c}g_3(c,a)^{*c})_{.c}F_3$ $\quad h(C_{g_3^2}(\overline{E})) = (a_{.c}g(c,a)^{*c})_{.c}F$
$C_{f_4^1}(\overline{E}) = ((a_{.a}f_4(a)^{*a})_{.a}b)_{.b}F_3$ $\quad h(C_{f_4^1}(\overline{E})) = ((a_{.a}f(a)^{*a})_{.a}b)_{.b}F$
$C_{h_5^1}(\overline{E}) = b_{.b}F_3$ $\quad h(C_{h_5^1}(\overline{E})) = b_{.b}F$

The set of states of the automaton C_E *is* $Q = \{(\varepsilon^1, C_{\varepsilon^1}(\overline{E})),$
$(f_1^1, C_{f_1^1}(\overline{E})), (h_2^1, C_{h_2^1}(\overline{E})), (g_3^1, C_{g_3^1}(\overline{E})), (g_3^2, C_{g_3^2}(\overline{E})), (f_4^1, C_{f_4^1}(\overline{E})), (h_5^1, C_{h_5^1}(\overline{E}))\}$.
The set of transition rules Δ *is*

$f((f_1^1, C_{f_1^1}(\overline{E}))) \to (\varepsilon^1, C_{\varepsilon^1}(\overline{E}))$
$g((g_3^1, C_{g_3^1}(\overline{E})), (g_3^2, C_{g_3^2}(\overline{E}))) \to (\varepsilon^1, C_{\varepsilon^1}(\overline{E}))$
$f((f_1^1, C_{f_1^1}(\overline{E}))) \to (f_1^1, C_{f_1^1}(\overline{E}))$
$h((h_2^1, C_{h_2^1}(\overline{E}))) \to (\varepsilon^1, C_{\varepsilon^1}(\overline{E}))$
$f((f_4^1, C_{f_4^1}(\overline{E}))) \to (f_4^1, C_{f_4^1}(\overline{E}))$
$h((h_2^1, C_{h_2^1}(\overline{E}))) \to (f_1^1, C_{f_1^1}(\overline{E}))$
$f((f_4^1, C_{f_4^1}(\overline{E}))) \to (g_3^1, C_{g_3^1}(\overline{E}))$
$g((g_3^1, C_{g_3^1}(\overline{E})), (g_3^2, C_{g_3^2}(\overline{E}))) \to (g_3^1, C_{g_3^1}(\overline{E}))$
$f((f_4^1, C_{f_4^1}(\overline{E}))) \to (\varepsilon^1, C_{\varepsilon^1}(\overline{E}))$
$h((h_5^1, C_{h_5^1}(\overline{E}))) \to (\varepsilon^1, C_{\varepsilon^1}(\overline{E}))$
$h((h_2^1, C_{h_2^1}(\overline{E}))) \to (h_2^1, C_{h_2^1}(\overline{E}))$
$f((f_1^1, C_{f_1^1}(\overline{E}))) \to (h_2^1, C_{h_2^1}(\overline{E}))$
$f((f_4^1, C_{f_4^1}(\overline{E}))) \to (h_5^1, C_{h_5^1}(\overline{E}))$
$h((h_5^1, C_{h_5^1}(\overline{E}))) \to (h_5^1, C_{h_5^1}(\overline{E}))$
$h((h_5^1, C_{h_5^1}(\overline{E}))) \to (f_4^1, C_{f_4^1}(\overline{E}))$
$h((h_5^1, C_{h_5^1}(\overline{E}))) \to (g_3^1, C_{g_3^1}(\overline{E}))$
$b \to (f_1^1, C_{f_1^1}(\overline{E}))$ $\quad$ $b \to (\varepsilon^1, C_{\varepsilon^1}(\overline{E}))$
$b \to (g_3^1, C_{g_3^1}(\overline{E}))$ $\quad$ $b \to (h_2^1, C_{h_2^1}(\overline{E}))$
$a \to (g_3^2, C_{g_3^2}(\overline{E}))$ $\quad$ $b \to (h_5^1, C_{h_5^1}(\overline{E}))$
$b \to (f_4^1, C_{f_4^1}(\overline{E}))$

The number of states is $|Q| = 5$ *and the number of transition rules is* $|\Delta| = 15$. *The k-C-Continuation Automaton associated with* E *is given in Figure 3.5.*

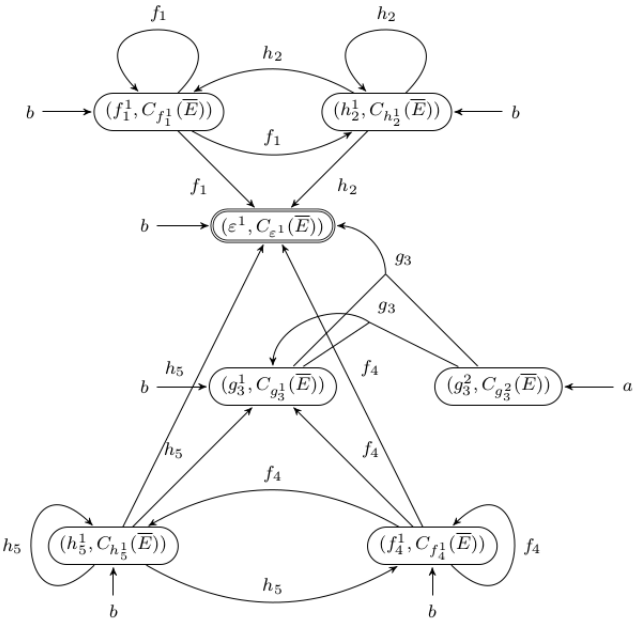


Figure 3.5: K-C-Continuation Automaton of Example 3.1.6 [MSZ17]

3.2 Part Two: From Tree Automata to Regular Tree Expressions

Unlike the first part, algorithms that converts tree automata to regular tree expressions have not been extensively studied. In this part, we present an algorithm that, for a given tree automaton $\mathcal{A}$, extracts a regular tree expression E such that E denotes the language accepted by $\mathcal{A}$.

In the case of words, there are a few classical algorithms for converting finite automata into equivalent regular expressions, namely: the McNaughton-Yamada algorithm [MY60], the algorithm based on Arden's lemma [Ard61, Con12], and the state elimination technique [BM63]. All these approaches are more or less reformulations of the same underlying algorithmic idea, and they yield almost the same regular expressions [GH15].

A well known approach to solve the conversion problem from finite automata to regular expressions is the algorithm based on Arden's lemma [Ard61, Con12]. It puts forward a set of language equations for a given finite automaton. Here, the i$^{\text{th}}$ equation describes the set X_i of words w such that the given automaton can go from the i$^{\text{th}}$ state to an accepting state on reading w. That system of equations can be resolved by eliminating X_i using a similar method to the Gaussian elimination [Atk89].

The McNaughton-Yamada algorithm [MY60] maintains a matrix with regular expression entries, where the rows and columns are the states of the given automaton. The iterative algorithm uses a ranking on the state set, and proceeds in n rounds, n is the number of states in the given automaton $\mathcal{A}$. From these expressions a regular expression describing $\mathcal{L}(\mathcal{A})$ is obtained [GH15].

The state elimination algorithm maintains an extended finite automaton, whose transitions are labeled with regular expressions, rather than alphabet symbols. For a given finite automaton, the state elimination technique deletes a state at each step and changes the transitions accordingly. This process continues until the automaton is reduced to the starting state, a final state, and the transition between them. The regular expression labeling the transition specifies exactly the language accepted by $\mathcal{A}$ [Woo87].

Another approach for regular expression generation from word automata that haven't been quite investigated is the one using integrals. It has been proposed by Smith and Yau in [SY72]. The notion of regular expression integral is the inverse of of derivatives. The authors have shown how, from a given deterministic automaton, we can use the notion of an integral of a regular expression to determine its corresponding regular expression. The developed algorithm is claimed to be systematic and effective.

Hereafter, we will present the approach that has been generalized to trees: the construction of tree automata from a regular tree expression using Arden's lemma generalization [GMCZ15].

Arden's Lemma Generalization

The basic idea behind this construction is to extract an equation system from the tree automaton, where each state of this automaton is represented by regular expression denoting the language it accepts [Gue17]. Before presenting the tree generalization of Arden's lemma in [GMCZ15], the authors defined a set of properties on languages used throughout the construction.

The following lemma defines the tree language accepted by a state q in a tree automaton. This language is called the down language of q.

Lemma 3.2.1 *Let $A = (\Sigma, Q, Q_T, \Delta)$ be a FTA. Let $q \in Q$ be a state. Then:*

$$L(q) = \bigcup_{f(q_1,\ldots,q_n,q)\in\Delta} f(L(q_1),\ldots,L(q_n))$$

This lemma is used to define an equation system that can describe the relations between the down languages of the states of a given tree automaton.

Definition 3.2.2 *Let $\mathcal{A} = (\Sigma, Q, Q_T, \Delta)$ be a FTA with $Q = \{1, \ldots, n\}$. The equation system associated with $\mathcal{A}$ is the set of equations $\mathcal{X}_{\mathcal{A}}$ over the variables $E_1, \ldots, E_n$ defined by $\mathcal{X}_{\mathcal{A}} = \{\varepsilon_q | q \in Q\}$ where for any state $q \in Q$, ε_q is the equation $E_q = F_q$ with $F_q = \sum_{(f, q_1, \ldots, q_n, q) \in \Delta} f(E_{q_1}, \ldots, E_{q_n})$.*

Two systems over the same variables are equivalent if they admit the same solutions. Notice that a system does not necessarily admit a unique solution [GMCZ15].

Arden's lemma generalization to trees gives a solution of the recursive language equation $X = A.X \cup B$ where X is the unknown language.

Example 3.2.3 *Consider the automaton in Figure 3.6 taken from [GMCZ15]. The equation system associated with $\mathcal{A}$ is*

$$\mathcal{X} = \begin{cases} E_1 &= f(E_1, E_1) + f(E_2, E_4) \\ E_2 &= b + f(E_2, E_4) \\ E_3 &= a + h(E_4) \\ E_4 &= a + h(E_3) \end{cases}$$

By replacing E_4 in E_1, E_2 and E_3 we get:

$$\mathcal{X} = \begin{cases} E_1 &= f(E_1, E_1) + f(E_2, a + h(E_3)) \\ E_2 &= b + f(E_2, a + h(E_3)) \\ E_3 &= a + h(a + h(E_3)) \end{cases}$$

Using the split and factorization properties defined in [Gue17] on the third equation:

$$\mathcal{X} = \begin{cases} E_1 &= f(E_1, E_1) + f(E_2, a + h(E_3)) \\ E_2 &= b + f(E_2, a + h(E_3)) \\ E_3 &= (h(a + h(x_3)))^{x_3}_{.x_3} a \end{cases}$$

Then, by substituting E_3 in E_1 and E_2:

$$\mathcal{X} = \begin{cases} E_1 &= f(E_1, E_1) + f(E_2, a + h((h(a + h(x_3)))^{x_3}_{.x_3} a)) \\ E_2 &= b + f(E_2, a + h((h(a + h(x_3)))^{x_3}_{.x_3} a)) \end{cases}$$

Reapplying the same steps, we get by the end the following solutions:

$$\mathcal{X} = \begin{cases} E_1 &= (f(x_1, x_1))^{x_1}_{.x_1} (f(((f(x_2, a + h((h(a + h(x_3)))^{x_3}_{.x_3} a)))^{x_2}_{.x_2} b, a + h((h(a + h(x_3)))^{x_3}_{.x_3} a))) \\ E_2 &= ((f(x, a + h((h(a + h(x)))^{x_3}_{.x_3} a)))^{x_2}_{.x_2} b \\ E_3 &= (h(a + h(x_3)))^{x_3}_{.x_3} a \\ E_4 &= a + h((h(a + h(x_3)))^{x_3}_{.x_3} a) \end{cases}$$

As states 1 and 3 are the final ones, E_1 and E_3 are the solutions of the equation system $\mathcal{X}$. That is to say, the language denoted by the automaton $\mathcal{A}$ is:

$(f(x_1,x_1))^{x_1}_{.x_1}(f(((f(x_2,a+h((h(a+h(x_3)))^{x_3}_{.x_3}a))))^{x_2}_{.x_2}b,a+h((h(a+h(x_3)))^{x_3}_{.x_3}a)))+(h(a+h(x_3)))^{x_3}_{.x_3}a$

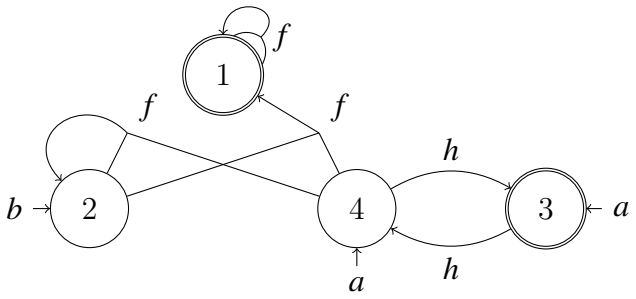


Figure 3.6: A tree automata example [Gue17]

3.3 Complexities and Morphic Links

Since there exist many algorithms converting regular expressions to automata, we have to study the relation between these automata and their complexities in order to determine which ones are optimal in space and time. Two automata are either isomorphic, i.e. they have the same structure up to the renaming of states, or an automaton is a quotient of the other, that is, it can be reduced or minimized to the other automaton.

In the case of strings, it has been shown that the position automaton $\mathcal{P}$ and the C-Continuations one are isomorphic [CZ01, CZ02]. Also, Ilie et al. [IY03] showed that the follow automaton $\mathcal{F}$ is a quotient of position automaton $\mathcal{P}$. Further, Ziadi et al. [CZ02] have shown that the equation automaton $\mathcal{E}$ is a quotient of $\mathcal{P}$. For trees, Mignot et al. [MSZ17] have proven that these morphic links are still valid for tree automata. They have also proposed a smaller automaton using García et al. [GLRÁ11] method for words which they denoted $\equiv_V$-NFA.

Figure 3.7 shows the links between the different automata constructed from a regular tree expression, where $\sim_e$, $\sim_f$ and $\equiv$ are the equivalence relations defined over the set of states of the K-C-Continuation automaton $\mathcal{C}$.

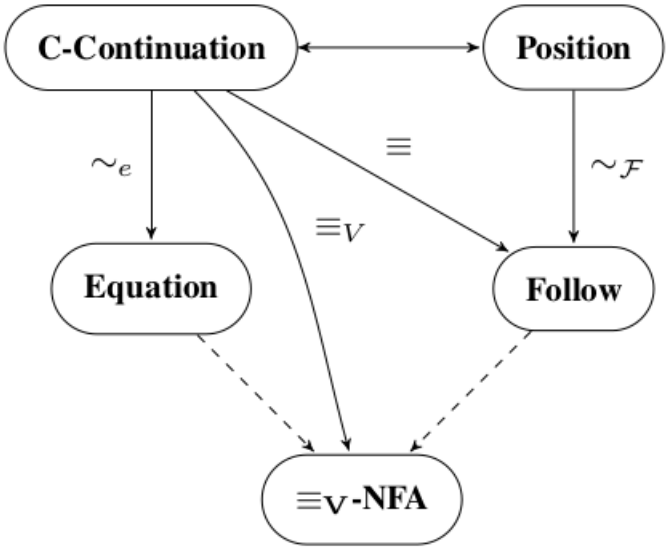


Figure 3.7: Morphic links between tree automata [MSZ14b].

The table given hereafter summarizes the different complexities of conversions from regular tree expressions to tree automata. Where R is the maximal rank appearing in the ranked alphabet, $|E|$ is the size of the regular tree expression E, $||E||$ is the alphabetic width of E and $|Q|$ is the number of states of the produced automaton.

Construction	Complexity
Equation tree automaton [KM11]	$\mathcal{O}(R.\|E\|^2)$
K-position tree automaton [LSZ13]	$\mathcal{O}(\|\|E\|\|.\|E\|)$
K-C-Continuations [MSZ14a]	$\mathcal{O}(\|Q\|.\|E\|)$
Follow tree automaton [MSZ17]	Not defined

3.4 Conclusion

In this chapter we have presented different proofs of Kleene theorem for trees: conversion from tree automata to regular tree expression and back.

We have started by giving a glimpse of the underlying theory in the word case with the most important references. Then, we have presented the different generalizations to trees for each approach.

Our contributions related to this part are the proposition of two algorithms: one for converting regular expressions into tree automata which is the generalization of Thompson's approach in words [BCZC15]. The other algorithm is the inverse operation using integrals [GBC15]. These two algorithms will be presented separately in the next two chapters.

Chapter 4

Construction of Tree Automata Using Thompson Generalization

In this chapter we present our first contribution; the generalization of Thompson automata to tree. We propose to construct a special tree automaton that can be viewed as a generalization of Thompson one for strings. A similar method constructing an automaton from a regular tree expression has been described in [PJM11], in which the authors use rather a different sort of tree automata (pushdown) and that requires a linearization of the trees involved; that is, the automata used there do not directly process trees but instead an encoding of trees into strings (postfix notation).

4.1 Recall of Thompson Automaton for Words

Regular expressions are notations for describing patterns of text invented by S. Kleene in the mid-1950s as a notation for finite automata. In fact, they have been proven to be equivalent to finite automata in what they represent. These expressions first appeared in a program setting in Ken Thompson's version of the QED text editor in the mid-1960s. In 1971, a patent was granted to Thompson for a mechanism for rapid text matching based on regular expressions. The text matching algorithm first checks regular expression for syntax, then it transforms the regular expression into postfix notation. After that, using a pushdown stack, it constructs a nondeterministic finite automaton for the regular expression. The nodes in this automaton are fragments of machine code [Cox11].

Thompson's original matcher was very fast because it combined two independent ideas. One was to generate machine instructions on the fly during matching so that it ran at machine speed rather than by interpretation. The other was to carry forward all possible matches at each stage, so it did not have to backtrack to look for alternative potential matches [OW07].

In his paper, Regular expression search algorithm [Tho68], Ken Thompson presents a method that allows the construction of nondeterministic finite word automata inductively from a given regular expression. The resulting automaton is composed of multiple automata constructed depending on the basic regular expression's operations. Figures 4.1 to 4.6 show how these fragments of automata are assembled according to the underlying operation [Bel14].

Let E and F be two regular expressions:

1. Figure 4.1 presents Thompson automaton for the empty language.

2. Thompson automaton for the empty word ε is presented in Figure 4.2.

3. Thompson automaton recognizing a letter $a \in \Sigma$ (Figure 4.3) is constructed by adding a transition labeled by a from the initial state to the final one.

4. Thompson automaton for the regular expression $E + F$ is constructed by adding two new states: an initial state linked to both initial states of E and F by ε, and a final state linked to both final states of E and F also by ε (Figure 4.4).

5. The construction of Thompson automaton for the concatenation operation $E.F$ is obtained by merging the final state of E with the initial state of F. The initial state of E becomes the initial state of the whole automaton $E.F$, and the final state of F becomes its final state (Figure 4.5).

6. Figure 4.6 presents the construction of Thompson automaton of the closure operation. An initial state and a final one are added with four ε-transitions:

   - a transition linking the initial state of E^* to E,
   - another transition from the final state of E to E^*,
   - a transition from the initial state of E to its final state in order to recognize the empty word,
   - and a final transition linking the final state of E to its initial state.

Figure 4.1: Thompson automaton of the empty language.

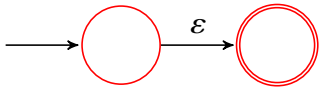


Figure 4.2: Thompson automaton of the empty word.

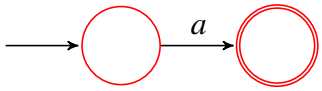


Figure 4.3: Thompson automaton of a letter a.

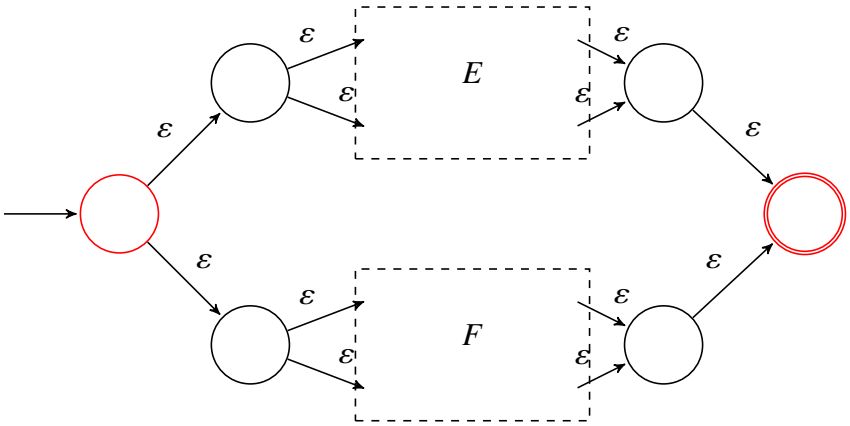


Figure 4.4: Thompson automaton of the union operation $E + F$.

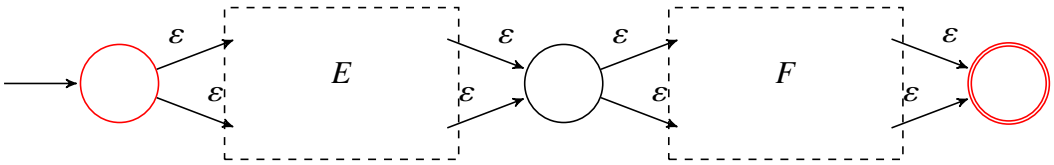


Figure 4.5: Thompson automaton of the concatenation operation $E.F$.

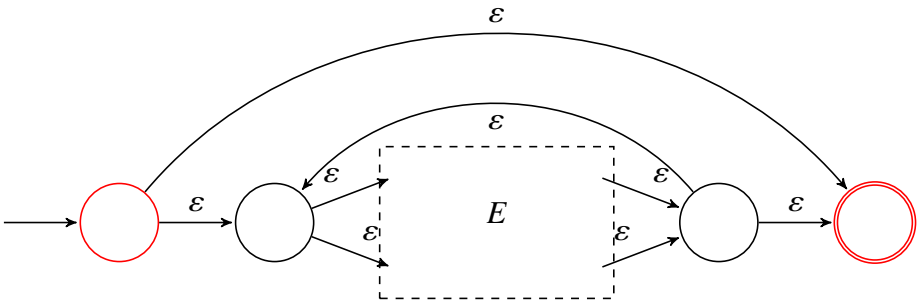


Figure 4.6: Thompson automaton of the closure operation E^*.

4.1.1 Example of a Thompson Word Automaton

Figure 4.7 shows the Thompson automaton constructed for the regular expression $(a + b)^* b(\varepsilon + a)(a + b)^*$. The automaton has 21 states and 27 transitions. Note that most transitions are labeled by the empty word ε.

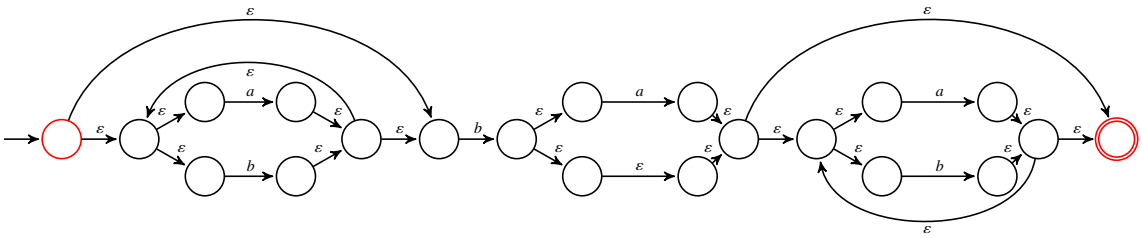


Figure 4.7: Example of a Thompson word automaton [Ber05].

4.1.2 Properties of Thompson Word Automata

For a regular expression E, the obtained automaton $\mathcal{A}$ using Thompson's construction has the following properties:

- $\mathcal{A}$ has exactly one initial state q_0 that is not accessible from any other state. That is, for any state q and any symbol a, $q_0 \notin \delta(q,a)$.
- $\mathcal{A}$ has exactly one final state q_f that is not co-accessible from any other state. That is, for any symbol f, $\delta(q_f,a) = \emptyset$.
- The number of states of $\mathcal{A}$ is linear in the size of E. That is $2|E| - c$, where c is the number of concatenation operations.
- The number of transitions from any state is at most two.
- Pattern matchers using Thompson's automaton require $O(|\mathcal{A}|m)$ time complexity, where $|\mathcal{A}|$ is the size of Thompson automaton and m is the size of the object text.

4.2 Generalization of Thompson Automata to Trees

By analogy to strings, we have proposed a construction of a bottom-up tree automaton from a regular tree expression that will be used later in tree pattern matching. To do this, we should first define the general form of such automaton that has nearly the same properties as Thompson automaton for words 4.1.2.

This work has been initiated in [Bel14], then presented in [BCZC15] and published as part of [BCCZ18].

General Form of a Thompson Tree Automaton

As mentioned before, the principle of this algorithm consists of building, by induction, an automaton from a regular tree expression. This construction has been used to propose a tree pattern matching algorithm that will be presented later when dealing with tree pattern matching problem (Chapter 6).

Before presenting the proposed construction, we introduce some notations and definitions. We add a symbol ε of rank 1 to the alphabet Σ. This symbol has the same meaning as the empty word in the case of strings. For example, the tree $f(\varepsilon(\varepsilon(a)),b)$ is equal to $f(a,b)$.

Definition 4.2.1 *Let t be a tree, we define the function ε-closure(t) that removes ε-nodes from t as follows:*

$$\begin{aligned} \varepsilon\text{-closure}(a) &= a \quad \textit{for each } a \textit{ in } \Sigma_0 \\ \varepsilon\text{-closure}(\varepsilon(t)) &= \varepsilon\text{-closure}(t) \\ \varepsilon\text{-closure}(g(t_1,\ldots,t_n)) &= g(\varepsilon\text{-closure}(t_1),\ldots,\varepsilon\text{-closure}(t_n)) \end{aligned}$$

From the definition of ε-closure(t) we can deduce the following properties:

Property 4.2.2 *Let t_1, $t_2 \in T_\Sigma$. We have then*

$$\varepsilon\text{-closure}(t_1 \cdot_c t_2) = \varepsilon\text{-closure}(t_1) \cdot_c \varepsilon\text{-closure}(t_2)$$

This property can be extended to sets of trees.

Property 4.2.3 *Let $s,t_1,\ldots,t_n \in T_\Sigma$. We have then*

$$\varepsilon\text{-closure}(s \cdot_c \{t_1,\ldots,t_n\}) = \varepsilon\text{-closure}(s) \cdot_c \{\varepsilon\text{-closure}(t_1),\ldots,\varepsilon\text{-closure}(t_n)\}$$

Given the difference between strings and trees concerning the concatenation and closure operations, we should take care when constructing a tree automaton. Indeed, we have designed a special form of tree automaton that allows us to inductively construct a tree automaton from a regular tree expression in a straightforward way. For the sake of simplicity we will use the name Thompson tree automaton to refer to this construction.

The basic idea of our construction is to build, from a given regular tree expression E, a finite bottom-up tree automaton which has the form illustrated by Figure 4.8. The main characteristic of this automaton is that it contains one initial state for each element of Σ_0 (the frame Q_{Σ_0}). This condition makes more sense for dealing with the concatenation operation, since as a result we have to perform concatenation in just one state.

In order to keep this form, some ε-transitions are added during the inductive constructions.

4.3 Constructing Thompson Tree Automata

Let E be a regular tree expression. The generalized Thompson automaton $\mathrm{Th}_E = (Q^E, \Sigma\cup\{\varepsilon\}, \{q^E\}, \Delta^E)$ over the alphabet $\Sigma\cup\{\varepsilon\}$ associated with E is defined inductively as follows.

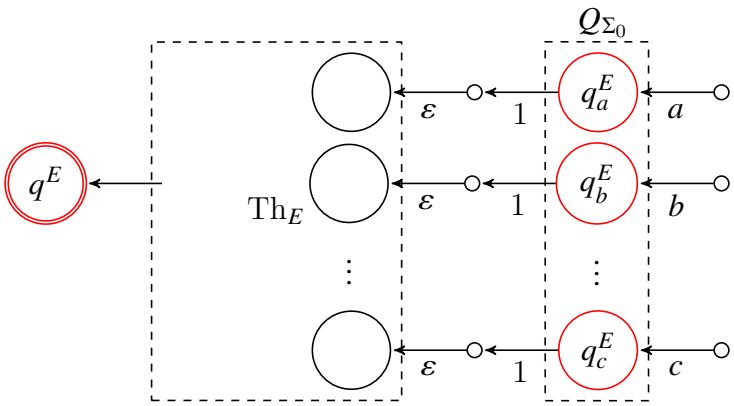


Figure 4.8: General form of Thompson tree automaton

Let Th_F, Th_G and Th_{E_i}, be the generalized Thompson automaton associated respectively with the tree expressions F, G and E_i, for $i = 1 \dots n$, then:

The empty language $E = 0$: Thompson's tree automaton defining the empty language contains only one state and no transitions (Figure 4.9).

So, $Q^E = \{q^E\}$ and $\Delta_E = \emptyset$

Figure 4.9: The empty language Thompson tree automaton

The leaf tree $E = a$, $a \in \Sigma_0$: Thompson's tree automaton recognizing a leaf tree language contains one state and one transition (Figure 4.10).

$Q^E = \{q^E\}$ and $\Delta_E = \{a \to q^E\}$

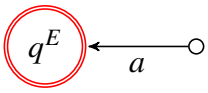


Figure 4.10: The leaf tree Thompson automaton

For the purpose of clarity, we refer hereafter to the set $\Delta^X \setminus \{a \to q_a^X \mid a \in \Sigma_0\}$ by $\Delta^{>0,X}$, where X is a regular tree expression.

The arity function $E = f(E_1, \dots, E_n)$: Thompson's tree automaton for the arity function for n regular tree expressions is constructed by adding new initial states for each automaton representing E_i and then, adding a transition with rank n, labeled by f, connecting the different E_i (Figure 4.11).

$$Q^E = \bigcup_{i=1}^{n} Q^{E_i} \cup \{q^E\} \cup \{q_a^E \mid a \in \Sigma_0\} \text{ and}$$

$$\Delta^E = \bigcup_{i=1}^{n} (\Delta^{>0,E_i}) \cup \{a \to q_a^E \mid a \in \Sigma_0\}$$
$$\cup \{f(q^{E_1}, q^{E_2}, \ldots, q^{E_n}) \to q^E\} \cup \{\varepsilon(q_a^E) \to q_a^{E_i} \mid a \in \Sigma_0, i = 1 \ldots n\}$$

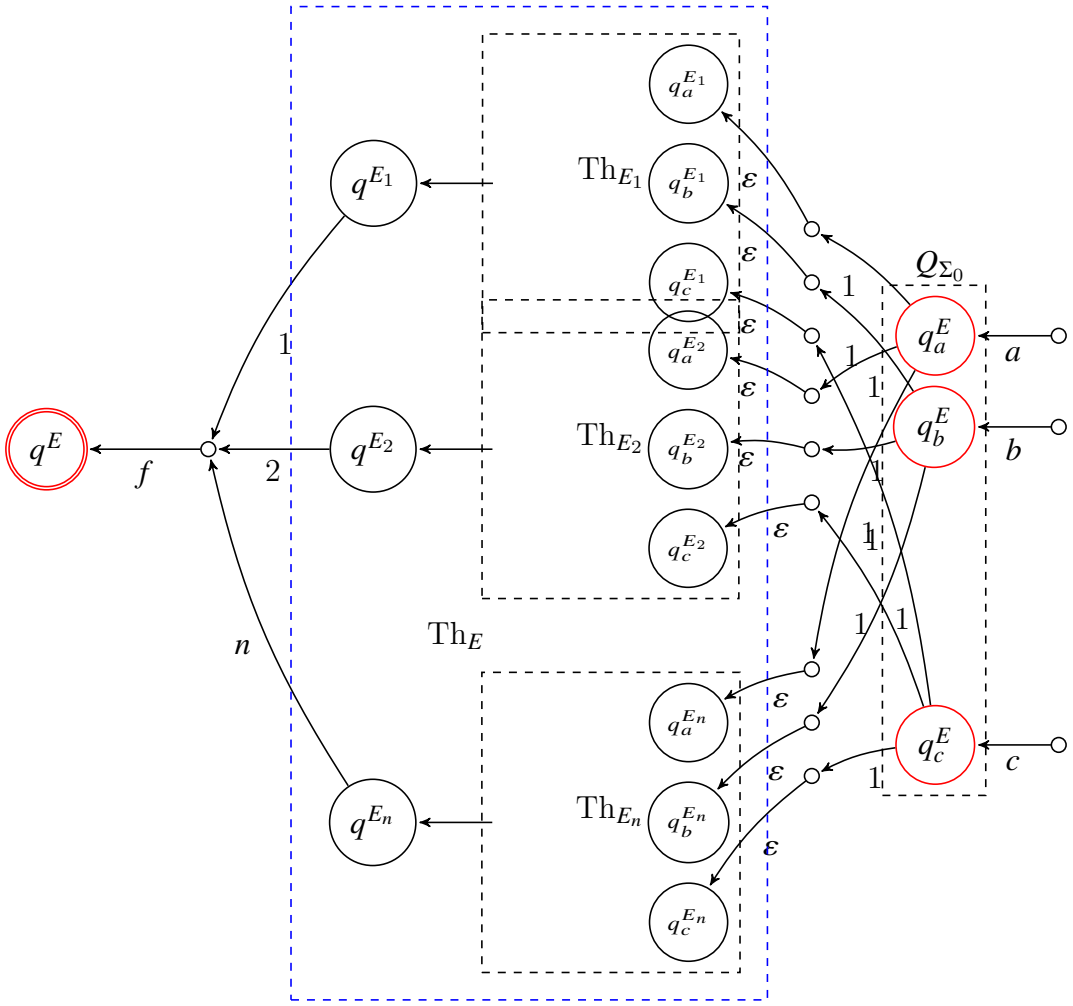


Figure 4.11: Thompson tree automaton for $E = f(E_1, E_2, .., E_n)$

The union function $E = F + G$: Thompson's tree automaton for the union operation is constructed like the arity function by adding new initial states for automata F and G, and then linking their initial states with ε-transitions to the initial state of E (Figure 4.12).

$$Q^E = Q^F \cup Q^G \cup \{q^E\} \cup \{q_a^E \mid a \in \Sigma_0\} \text{ and}$$

$$\Delta^E = \Delta^{>0,F} \cup \Delta^{>0,G} \cup \{a \to q_a^E \mid a \in \Sigma_0\}$$
$$\cup \{\varepsilon(q_a^E) \to q_a^F \mid a \in \Sigma_0\} \cup \{\varepsilon(q_a^E) \to q_a^G \mid a \in \Sigma_0\}$$
$$\cup \{\varepsilon(q^F) \to q^E\} \cup \{\varepsilon(q^G) \to q^E\}$$

The concatenation operation $E = F \cdot_c G$: Thompson's tree automaton for the concatenation operation is constructed by adding new initial states for F and G and

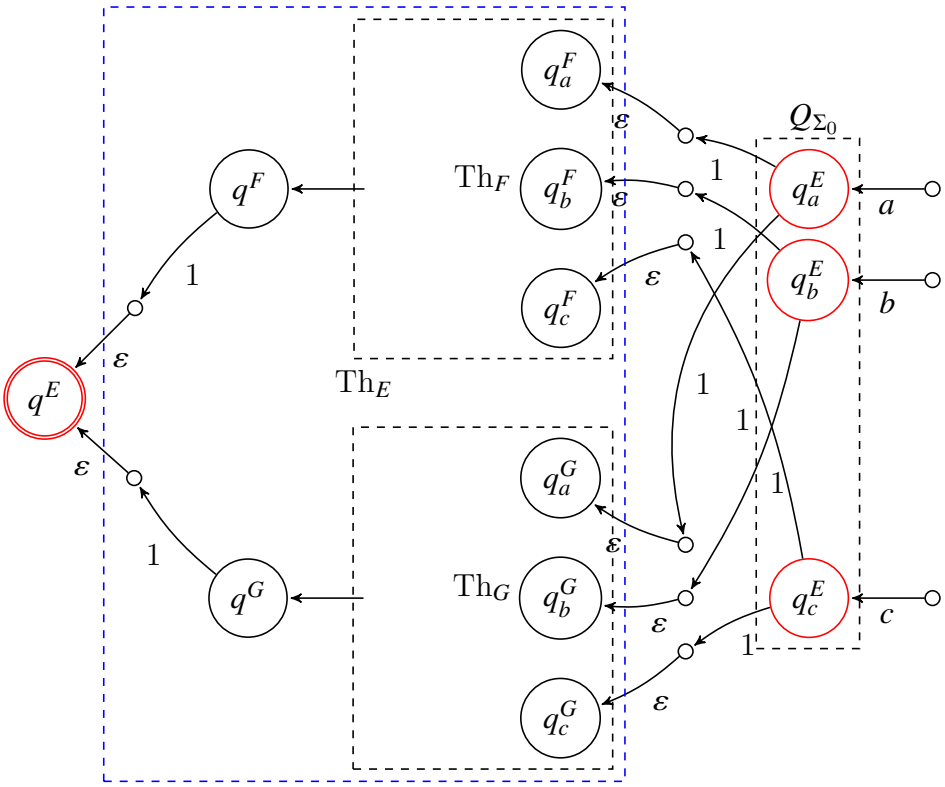


Figure 4.12: Thompson tree automaton for $E = F + G$

then adding an ε-transition from G to F. The final state of F will be the final one of the whole automaton (Figure 4.13).

$Q^E = Q^F \cup Q^G \cup \{q^E\} \cup \{q_a^E \mid a \in \Sigma_0\}$ and

$$\begin{aligned}\Delta^E =& \Delta^{>0,F} \cup \Delta^{>0,G} \cup \{a \to q_a^E \mid a \in \Sigma_0\} \\ & \cup \{\varepsilon(q_a^E) \to q_a^G \mid a \in \Sigma_0\} \cup \{\varepsilon(q_a^E) \to q_a^F \mid a \in \Sigma_0 \setminus \{c\}\} \\ & \cup \{\varepsilon(q^G) \to q_c^F\} \cup \{\varepsilon(q^F) \to q^E\}\end{aligned}$$

The closure operation $E = F^{*_c}$: Thompson's tree automaton for the closure operation is constructed by adding new initial states for E and then adding three ε-transitions in order to recognize the empty language and repetitions(Figure 4.14).

$Q^E = Q^F \cup \{q^E\} \cup \{q_a^E \mid a \in \Sigma_0\}$ and

$$\begin{aligned}\Delta^E =& \Delta^{>0,F} \cup \{a \to q_a^E \mid a \in \Sigma_0\} \\ & \cup \{\varepsilon(q_c^E) \to q^E\} \cup \{\varepsilon(q^F) \to q_c^F\} \cup \{\varepsilon(q^F) \to q^E\} \\ & \cup \{\varepsilon(q_a^E) \to q_a^F \mid a \in \Sigma_0\}\end{aligned}$$

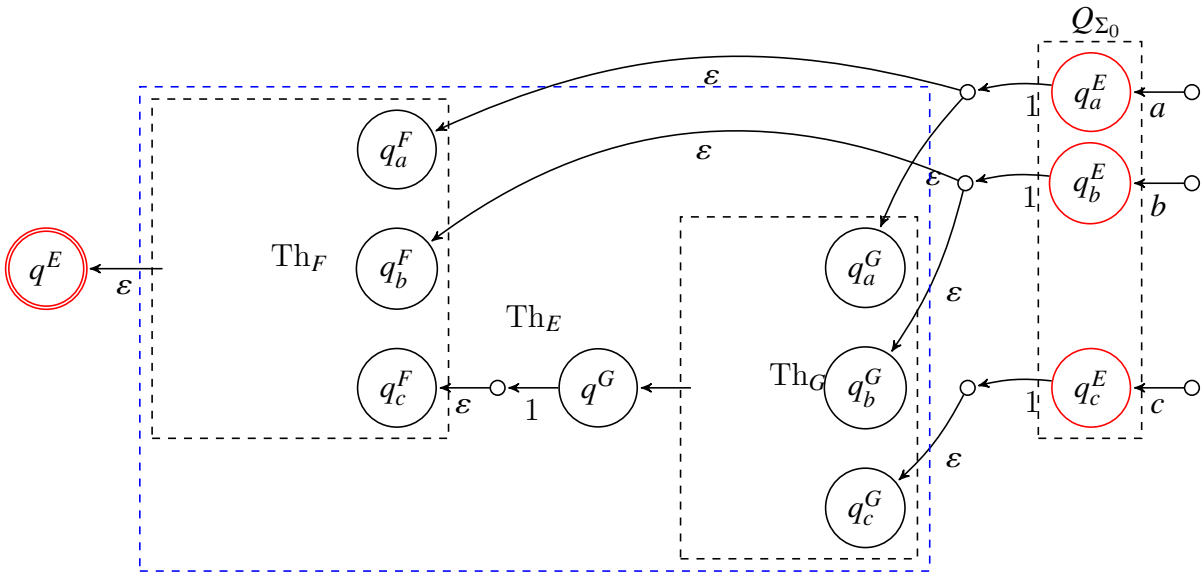


Figure 4.13: Thompson tree automaton for $E = F \cdot_c G$

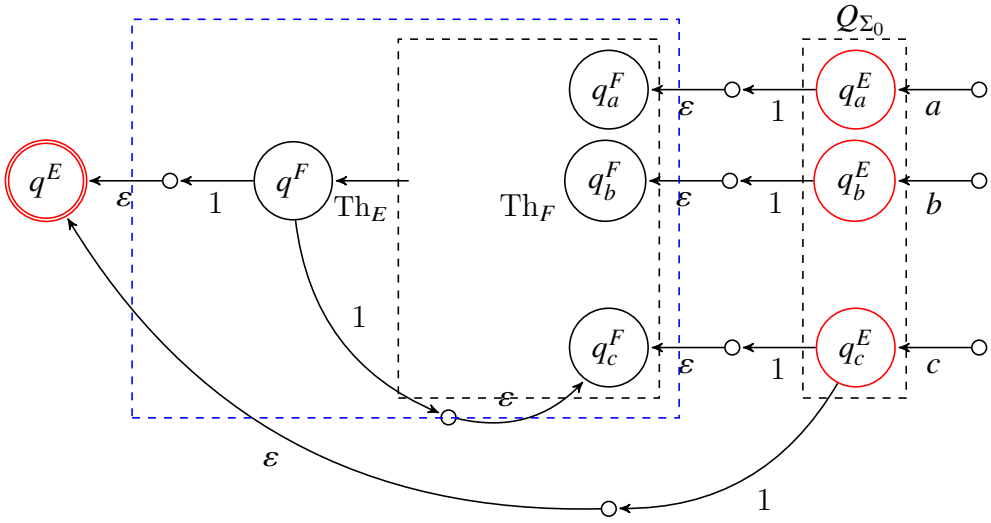


Figure 4.14: Thompson tree automaton for $E = F^{*,c}$

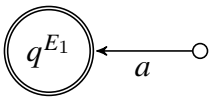
Before proving the validity of our generalization of Thompson tree automaton, that is: it recognizes the same language as E, we illustrate the process of construction by the following example.

4.4 Example of Constructing a Thompson Tree Automaton

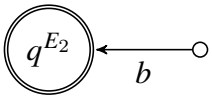
Let E be a regular tree expression such that $E = (f(a,b) + g(c) \cdot_c d)^{*,d}$. For the sake of simplicity, we relabel the regular tree expressions as follows:

$$\begin{aligned} E &= (f(a,b) + g(c) \cdot_c d)^{*,d} \\ &= (f(E_1,E_2) + g(c) \cdot_c d)^{*,d} \\ &= (f(E_1,E_2) + g(E_4) \cdot_c d)^{*,d} \\ &= (E_3 + E_5 \cdot_c E_6)^{*,d} \\ &= (E_3 + E_7)^{*,d} \\ &= (E_8)^{*,d} \end{aligned}$$

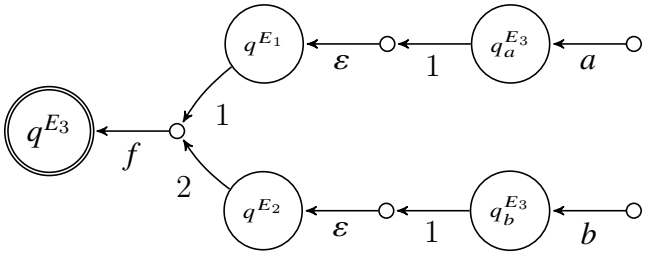
Figures 4.15–4.18 show the inductive construction of Thompson tree Automaton for E.



(a) Thompson tree automaton of $E_1 = a$

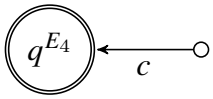


(b) Thompson tree automaton of $E_2 = b$

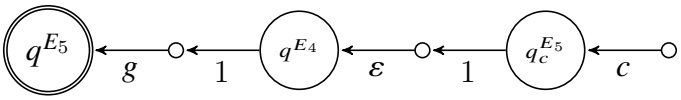


(c) Thompson tree automaton of $E_3 = f(a,b)$

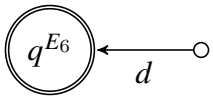
Figure 4.15: Thompson tree automaton of $f(a,b)$



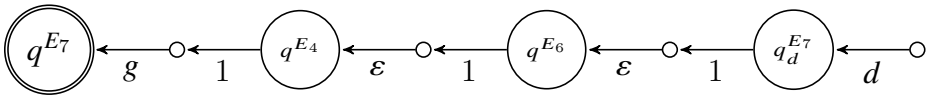
(a) Thompson tree automaton of $E_4 = c$



(b) Thompson tree automaton of $E_5 = g(c)$



(c) Thompson tree automaton for $E_6 = d$



(d) Thompson tree automaton of $E_7 = E_5 \cdot_c d$

Figure 4.16: Thompson tree automaton of $g(c) \cdot_c d$.

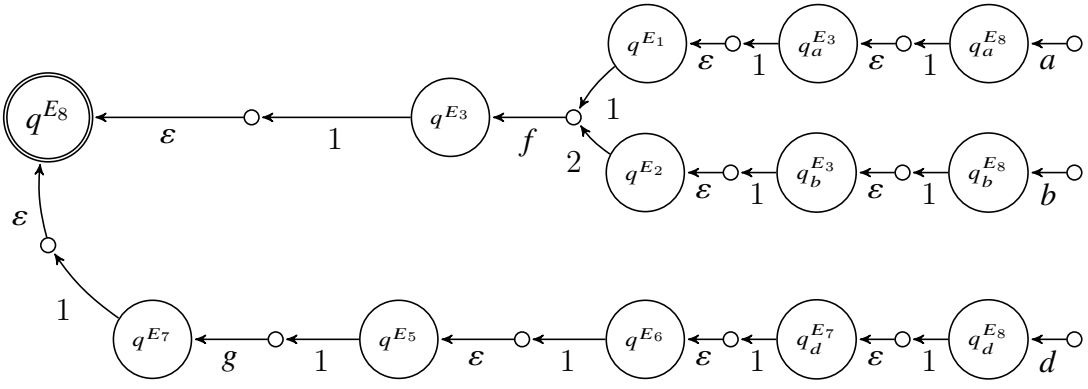


Figure 4.17: Thompson tree automaton of $E_8 = E_3 + E_7 = f(a,b) + g(c) \cdot_c d$

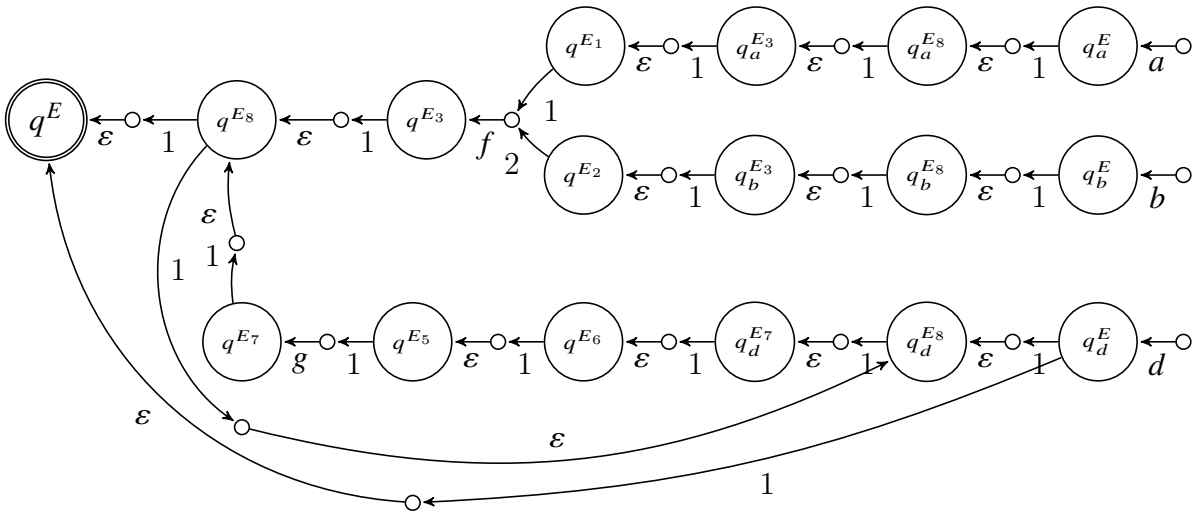


Figure 4.18: Thompson tree automaton of $E = E_8^{*,d} = (f(a,b) + g(c) \cdot_c d)^{*,d}$

The number of states of this Thompson automaton is $|Q| = 17$ and the number of transitions is $|\Delta| = 20$.

4.5 Proving the Validity of Thompson Tree Automata

Before using the constructed Thompson automaton we have to make sure that this automaton denotes the same tree language as the one recognized by the starting regular tree expression E. For this aim, we prove the following theorem.

Theorem 4.5.1 *For a regular tree expression E, ε-closure$(\mathcal{L}(\mathrm{Th}_E)) = [\![E]\!]$.*

In order to prove this theorem, which states that the language recognized by Th_E is equivalent to the one specified by the regular tree expression E, we prove the two next propositions. Let $\Sigma_0 = \{x_1, x_2, \cdots, x_n\}$ and $t, \bar{t}, \widetilde{t} \in T_\Sigma$ such that

$$\bar{t} = \Big(\cdots \Big((t \cdot_{x_1} \varepsilon(x_1)) \cdot_{x_2} \varepsilon(x_2) \Big) \cdots \Big) \cdot_{x_n} \varepsilon(x_n)$$

and

$$\widetilde{t} = \Big\{ \cdots \Big\{ \{ t\{\varepsilon(x_1) \leftarrow x_1\} \} \varepsilon(x_2) \leftarrow x_2 \Big\} \cdots \Big\} \varepsilon(x_n) \leftarrow x_n$$

$\bar{t}$ is the tree t where we have substituted all occurrences of x_i by the subtree $\varepsilon(x_i)$, and $\widetilde{t}$ is the tree t where we have done the inverse operation, i.e. subtrees $\varepsilon(x_i)$ are replaced by leaves x_i.

Let $\mathcal{A}$, $\overline{\mathcal{A}}$ and $\widetilde{\mathcal{A}}$ be tree automata such that $\mathcal{A} = (Q, \Sigma, q^{\mathcal{A}}, \Delta), \overline{\mathcal{A}} = (\overline{Q}, \Sigma, q^{\overline{\mathcal{A}}}, \overline{\Delta})$ and $\widetilde{\mathcal{A}} = (\widetilde{Q}, \Sigma, q^{\widetilde{\mathcal{A}}}, \widetilde{\Delta})$, where

$$
\begin{aligned}
\overline{Q} &= Q \cup \{q'^{\mathcal{A}}_a \mid \forall\, a \in \Sigma_0 \wedge q^{\mathcal{A}}_a \in Q\} \\
q^{\overline{\mathcal{A}}} &= q^{\mathcal{A}} \\
\overline{\Delta} &= \Delta \setminus \{a \to q^{\mathcal{A}}_a \mid a \in \Sigma_0 \wedge q^{\mathcal{A}}_a \in Q\} \\
&\quad \cup \{a \to q'^{\mathcal{A}}_a \mid a \in \Sigma_0 \wedge q'^{\mathcal{A}}_a \in \overline{Q}\} \\
&\quad \cup \{\varepsilon(q'^{\mathcal{A}}_a) \to q^{\mathcal{A}}_a\}
\end{aligned}
$$

and

$$
\begin{aligned}
\widetilde{Q} &= Q \setminus \{q'^{\mathcal{A}}_a, \mid \forall\, a \in \Sigma_0 \wedge q'^{\mathcal{A}}_a \in Q\} \\
q^{\widetilde{\mathcal{A}}} &= q^{\mathcal{A}} \\
\widetilde{\Delta} &= \Delta \setminus \{a \to q'^{\mathcal{A}}_a \mid a \in \Sigma_0 \wedge q'^{\mathcal{A}}_a \in Q\} \\
&\quad \setminus \{\varepsilon(q'^{\mathcal{A}}_a) \to q^{\mathcal{A}}_a \mid q^{\mathcal{A}}_a \in Q\} \\
&\quad \cup \{a \to q^{\mathcal{A}}_a \mid a \in \Sigma_0 \wedge q^{\mathcal{A}}_a \in Q\}
\end{aligned}
$$

These automata are introduced in order to improve the readability of the proofs of Theorem 4.5.1. $\overline{\mathcal{A}}$ is the automaton to which we have added ε-transitions after initial transitions, and $\widetilde{\mathcal{A}}$ is the one from which we have removed these ε-transitions.

Proposition 4.5.2 *If $t \in \mathcal{L}(\mathcal{A})$, then $\bar{t} \in \mathcal{L}(\overline{\mathcal{A}})$.*

Proof. We have $t \in \mathcal{L}(\mathcal{A})$, so $\Delta^*_{\mathcal{A}}(t) = q^{\mathcal{A}}$. According to the construction of $\overline{\mathcal{A}}$, every transition of the form $a \to q^{\mathcal{A}}_a$ such that $a \in \Sigma_0 \wedge q^{\mathcal{A}}_a \in Q$ in the path recognizing t in $\mathcal{A}$ is replaced by the two transitions $a \to q'^{\mathcal{A}}_a$ and $\varepsilon(q'^{\mathcal{A}}_a) \to q^{\mathcal{A}}_a$ with $a \in \Sigma_0 \wedge q'^{\mathcal{A}}_a \in \overline{Q}$. Then $\overline{\Delta}^*(\bar{t}) = q^{\mathcal{A}} = q^{\overline{\mathcal{A}}}$, that is $\bar{t} \in \mathcal{L}(\overline{\mathcal{A}})$. □

Remark 4.5.3 *Using Property 4.2.2, it is obvious that ε-closure$(t) = \varepsilon$-closure$(\bar{t})$.*

$$
\begin{aligned}
\varepsilon\text{-closure}(\bar{t}) &= \varepsilon.\text{-closure}\Big(\Big(\big((t \cdot_{x_1} \varepsilon(x_1)) \cdot_{x_2} \varepsilon(x_2)\big) \cdots\Big) \cdot_{x_n} \varepsilon(x_n)\Big) \\
&= \Big(\cdots\Big(\big(\varepsilon\text{-closure}(t) \cdot_{x_1} \varepsilon\text{-closure}(\varepsilon(x_1))\big) \cdot_{x_2} \varepsilon\text{-closure}(\varepsilon(x_2))\Big) \\
&\qquad \cdots\Big) \cdot_{x_n} \varepsilon\text{-closure}(\varepsilon(x_n)) \\
&= \Big(\cdots\big((\varepsilon\text{-closure}(t) \cdot_{x_1} (x_1)) \cdot_{x_2} (x_2)\big) \cdots\Big) \cdot_{x_n} (x_n) \\
&= \varepsilon\text{-closure}(t)
\end{aligned}
$$

Proposition 4.5.4 *If $t \in \mathcal{L}(\mathcal{A})$, then $\widetilde{t} \in \mathcal{L}(\widetilde{\mathcal{A}})$.*

Proof.

We have $t \in \mathcal{L}(\mathcal{A})$, that is $\Delta^*_{\mathcal{A}}(t) = q^{\mathcal{A}}$. According to the construction of the automaton $\widetilde{\mathcal{A}}$, in the path recognizing t in $\mathcal{A}$ we substitute all transitions of the form $a \to {q'}_a^{\mathcal{A}}$ and $\varepsilon({q'}_a^{\mathcal{A}}) \to q_a^{\mathcal{A}}$ with $a \in \Sigma_0 \wedge {q'}_a^{\mathcal{A}} \in Q$ by the transition $a \to q_a^{\mathcal{A}}$ for each $a \in \Sigma_0 \wedge q_a^{\mathcal{A}} \in Q$. Then $\widetilde{\Delta}^*(\widetilde{t}) = q^{\mathcal{A}} = q^{\widetilde{\mathcal{A}}}$, that is $\widetilde{t} \in \mathcal{L}(\widetilde{\mathcal{A}})$. □

Remark 4.5.5 *Like Remark 4.5.3, it is clear that* ε-closure$(t) = \varepsilon$-closure$(\widetilde{t})$.

In order to prove Theorem 4.5.1, we prove the two next lemmas which verify for a tree $t \in [\![E]\!]$ that there exists an equivalent tree $t' \in \mathcal{L}(\mathrm{Th}_E)$ with ε-transitions. This is accomplished by induction on the structure of the tree automaton using Propositions 4.5.2 and 4.5.4. We give the proof for the cases of a leaf tree, arity and concatenation operations. The case of union is straightforward and the closure is similar to the concatenation.

Lemma 4.5.6 *For each tree* $t \in [\![E]\!]$*, there exists a tree* $t' \in \mathcal{L}(\mathrm{Th}_E)$ *such that* ε-closure$(t') = t$.

Proof.

Case $E = a$

Let $t \in [\![E]\!]$, $t = a$. From Thompson leaf tree automaton construction, we have $\mathcal{L}(\mathrm{Th}_E) = \{a\}$, so $t' = a$. We have also ε-closure$(t') = \varepsilon$-closure$(a) = t$.

Case $E = g(E_1, E_2, \ldots, E_n)$ where n is the rank of g.

Let $t \in [\![E]\!]$, so $t = g(e_1, e_2, \ldots, e_n)$, with $t_i \in [\![E_i]\!]$ and $1 < i \leqslant n$. According to the induction hypothesis, there exist $t'_i \in \mathcal{L}(\mathrm{Th}_{E_i})$ such that ε-closure$(t'_i) = t_i$ with $i = 1 \ldots n$. We assume that $\bar{t}_i = {t'}_i$ since according to the construction of Thompson automaton generalisation, t'_i has the same structure as $\bar{t}_i$. This assumption will be used in the concatenation case as well.

According to the construction of Thompson automaton for the arity, and using Proposition 4.5.2, we have $\overline{\Delta}^*(\overline{t_i}) = q^{E_i}$ for $i = 1 \cdots n$. Moreover, we have $g(q^{E_1}, q^{E_2}, \ldots, q^{E_n}) \to q^E \in \Delta$, then $\Delta^*(\bar{t}) = q^E$ where $\bar{t} = g(\overline{t_1}, \overline{t_2}, \cdots, \overline{t_n})$, that is $\bar{t} \in \mathcal{L}(\mathrm{Th}_E)$. We show that ε-closure$(\bar{t}) = t$. From Remark 4.5.3, we have ε-closure$(\overline{t_i}) = \varepsilon$-closure$(t_i)$, so

$$\varepsilon\text{-closure}(\bar{t}) = \varepsilon\text{-closure}(g(\overline{t_1}, \overline{t_2}, \cdots, \overline{t_n}))$$

Using Property 4.2.2, we get

$$\varepsilon\text{-closure}(\bar{t}) = g(\varepsilon\text{-closure}(\overline{t_1}), \varepsilon\text{-closure}(\overline{t_2}), \cdots, \varepsilon\text{-closure}(\overline{t_n}))$$

Using Remark 4.5.3, we have

$$\varepsilon\text{-closure}(\bar{t}) = g(\varepsilon\text{-closure}(t_1), \varepsilon\text{-closure}(t_2), \cdots, \varepsilon\text{-closure}(t_n))$$

Using the induction hypothesis, we get

$$\varepsilon\text{-closure}(\bar{t}) = g(t_1, t_2, \cdots, t_n) = t$$

Case $E = F \cdot_c G$

Let $t \in [\![E]\!]$, $t \in [\![F]\!] \cdot_c [\![G]\!]$ means $t \in \{(t_f) \cdot_c \{t_1, \ldots, t_k\}\}$, such that $t_i \in [\![G]\!]$, $i = 1 \ldots k$ and $t_f \in [\![F]\!]$. According to the induction hypothesis, there exist $t'_f, t'_1, \ldots, t'_k$ with $t'_f \in \mathcal{L}(\mathrm{Th}_F)$ and $t'_i \in \mathcal{L}(\mathrm{Th}_G)$, $i = 1 \ldots k$, such that $\varepsilon\text{-closure}(t'_f) = t_f$ and $\varepsilon\text{-closure}(t'_k) = t_k$. Let $\bar{t}_i = t'_i$, $\bar{t}_f = t'_f$ and $\bar{t} \in \{(\bar{t}_f) \cdot_c \{\bar{t}_1, \bar{t}_2, \cdots, \bar{t}_k\}\}$. According to the construction of Thompson automaton of concatenation and using Proposition 4.5.2, we have

$$\Delta^*(\overline{t_i}) = q^G \tag{4.1}$$

According to the same construction and also using Proposition 4.5.2, the same transitions are replaced except for $a = c$ where c is the concatenation symbol, which is replaced by $\varepsilon(q^G) \to q^F_c$. Then, we have

$$\Delta^*(\overline{t_f}) = q^F. \tag{4.2}$$

From 4.1 and 4.2 we get $\Delta^*(\bar{t}) = q^E$. We show that $\varepsilon\text{-closure}(\bar{t}) = t$.

$\varepsilon\text{-closure}(\bar{t}) = \varepsilon\text{-closure}(\{(\bar{t}_f \cdot)_c \{\bar{t}_1, \bar{t}_2, \cdots, \bar{t}_k\}\})$

Using Property 4.2.3, we get

$$\varepsilon\text{-closure}(\bar{t}) = \varepsilon\text{-closure}(\bar{t}_f) \cdot_c \{\varepsilon\text{-closure}(\bar{t}_1), \varepsilon\text{-closure}(\bar{t}_2), \cdots, \varepsilon\text{-closure}(\bar{t}_k)\}$$

Using Remark 4.5.3, we have

$$\varepsilon\text{-closure}(\bar{t}) = \varepsilon\text{-closure}(t_f) \cdot_c \{\varepsilon\text{-closure}(t_1), \varepsilon\text{-closure}(t_2), \cdots, \varepsilon\text{-closure}(t_k)\}$$

And finally using the induction hypothesis, we get

$$\varepsilon\text{-closure}(\bar{t}) = t_f \cdot_c \{t_1, t_2, \cdots, t_k\} = t$$

□

Lemma 4.5.7 *If* $t \in \mathcal{L}(\mathrm{Th}_E)$*, then* ε-closure$(t) \in [\![E]\!]$.

Proof.

Case of leaf tree automaton

Let $t \in \mathcal{L}(\mathrm{Th}_E)$. According to the construction of the generalized Thompson automaton of leaf tree we have $t = a$. Moreover, we have ε-closure$(t) = \varepsilon$-closure$(a) = a$ and $a \in [\![E]\!]$.

Case of arity automaton

Let $t \in \mathcal{L}(\mathrm{Th}_E)$, $t = g(t_1, t_2, \ldots, t_n)$. According to the induction hypothesis and using Proposition 4.5.4, there exist $t_1, t_2, \ldots, t_n \in T_\Sigma$ such that $\widetilde{t_1} \in \mathcal{L}(\mathrm{Th}_{E_1})$ and ε-closure$(\widetilde{t_k}) \in [\![E_k]\!]$, for $k = 1 \ldots n$.

Furthermore, we have

$$\varepsilon\text{-closure}(t) = \varepsilon\text{-closure}(g(t_1, t_2, \ldots, t_n))$$

From Definition 4.2.1, we have

$$\varepsilon\text{-closure}(t) = g(\varepsilon\text{-closure}(t_1), \varepsilon\text{-closure}(t_2), \ldots, \varepsilon\text{-closure}(t_n))$$

And using Remark 4.5.5, we get

$$\varepsilon\text{-closure}(t) = g(\varepsilon\text{-closure}(\widetilde{t_1}), \varepsilon\text{-closure}(\widetilde{t_2}), \ldots, \varepsilon\text{-closure}(\widetilde{t_n}))$$

Using the induction hypothesis, we have ε-closure$(\widetilde{t_i}) \in [\![E_i]\!]$, for $i = 1 \ldots n$ where n is the rank of g. Then,

$$g(\varepsilon\text{-closure}(\widetilde{t_1}), \varepsilon\text{-closure}(\widetilde{t_2}), \ldots, \varepsilon\text{-closure}(\widetilde{t_n})) \in [\![E]\!]$$

Case of concatenation automaton

Let $t \in \mathcal{L}(\mathrm{Th}_E)$, then $t \in \{(t_f) \cdot_c \{t_1, \ldots, t_k\}\}$, such that $t_i \in [\![G]\!]$, $i = 1 \ldots k$. According to the induction hypothesis and using Proposition 4.5.4, there exist $t_1, \ldots, t_k, t_f \in T_\Sigma$ such that $\widetilde{t_i} \in \mathcal{L}(\mathrm{Th}_G)$ with ε-closure$(\widetilde{t_i}) \in [\![G]\!]$, $i = 1 \ldots k$, and $\widetilde{t_f} \in \mathcal{L}(\mathrm{Th}_F)$ with ε-closure$(\widetilde{t_f}) \in [\![F]\!]$.

Moreover, we have

$$\varepsilon\text{-closure}(t) = \varepsilon\text{-closure}(t_{f \cdot_c} \{t_1, \ldots, t_k\})$$

Using Property 4.2.3, we get

$$\varepsilon\text{-closure}(t) = \varepsilon\text{-closure}(t_f) \cdot_c \{\varepsilon\text{-closure}(t_1), \ldots, \varepsilon\text{-closure}(t_k)\}$$

Using Remark 4.5.5, we have

$$\varepsilon\text{-closure}(t) = \varepsilon\text{-closure}(\widetilde{t_f}) \cdot_c \{\varepsilon\text{-closure}(\widetilde{t_1}), \ldots, \varepsilon\text{-closure}(\widetilde{t_k})\}$$

And finally using the induction hypothesis, we get

$$\varepsilon\text{-closure}(t) \in [\![F]\!] \cdot_c [\![G]\!] \in [\![F \cdot_c G]\!] \in [\![E]\!]$$

□

4.6 Properties of Thompson Tree Automaton

Let $q \in Q$ be a state. Let $Q^{\varepsilon}_{-}(q) = \{p \in Q \mid \varepsilon(p) \longrightarrow q\}$.

From the construction of Thompson tree automata, we deduce the following property.

Property 4.6.1 *For a state $q \in Q$, we have $|Q^{\varepsilon}_{-}(q)| \leqslant 2$.*

Corollary 4.6.2 *The number of transitions in the generalized Thompson tree automaton for a regular tree expression E is linear in $|E|$.*

Proof. As each symbol in the regular tree expression E generates a constant number of transitions in the inductive construction of Thompson tree automaton, and since we can consider each regular tree expression as a tree where leaves represent the symbols of the alphabet and internal nodes represent the regular tree expression operators, so the number of transitions generated for this automaton's construction is $O(|E|)$. □

4.7 Conclusion

In this chapter, we have presented our first contribution: the generalization of Thompson construction from words to trees. Among the powerful properties of this construction is the linear number of transitions w.r.t. the length of the starting regular tree expression.

This contribution will be used two chapters later when dealing with tree pattern matching problem. In fact, we have proposed a tree pattern matching algorithm based on Thompson tree automata as it has been done for words.

Chapter 5

Synthesis of Regular Tree Expressions Using Integrals

In this chapter, we present our generalization of the notion of integrals from regular string expressions to regular tree expressions. An earlier version of this contribution has been presented in [GBC15].

Integrals might be considered as the inverse operation of derivatives, so we also show the link between tree integrals and tree derivatives. This generalization is used to denote by a regular tree expression, the tree language accepted by a tree automaton.

5.1 Preliminaries

The notion of integrals was first introduced for words by Smith et al. [SY72] as the inverse operation of derivatives proposed by Brzozowski et al. [Brz64]. On trees, Kuske and Meinecke [KM11] have proposed for the first time the notion of partial derivation on tree languages. It consists of the languages $L_1, \ldots, L_n$ resulting from the removal of a root symbol f from all trees described by the regular tree expression $f(L_1, \ldots, L_n)$. As we deal with ordered trees, the result of such an operation is a tuple (or a set of tuples) of ordered regular tree expressions. Integrals on trees must then take as input a tuple of tree languages $L_1, \ldots, L_n$ and a symbol f, and outputs a new tree language corresponding to the composition of the tuples and f.

Before introducing integrals for trees, we start by recalling some basic properties of regular tree expressions in the following lemma.

Lemma 5.1.1 *Let E_1, E_2, E_3 be regular tree expressions over Σ. Let $c \in \Sigma_0$. Then, the following statements hold.*

$$E_1 + E_1 \equiv E_1 \tag{5.1}$$

$$E_1 + 0 \equiv E_1 \text{ (Identity element of } +) \tag{5.2}$$

$$E_1 + E_2 \equiv E_2 + E_1 \text{ (Commutativity of } +) \tag{5.3}$$

$$(E_1 + E_2).{}_c E_3 \equiv E_1.{}_c E_3 + E_2.{}_c E_3 \text{ (Right distributivity)} \tag{5.4}$$

$$E_1.{}_c E_1^{*c} \equiv E_1^{*c}.{}_c E_1 \equiv E_1.{}_c E_1^{*c} \text{ if } c \in [\![E_1]\!] \text{ and } E_1^{*c} \text{ otherwise} \tag{5.5}$$

5.2 Integral of Regular Tree Expressions

We define the concatenation operation on a tuple of regular tree expressions as follows:

Definition 5.2.1 *Let $\mathcal{N}$ be an n-tuple of regular tree expressions. Then the n-tuple concatenation with the regular tree expression F is defined as follows:*

$$\mathcal{N}._c F = (E_1._c F, \ldots, E_n._c F)$$

This definition is used later for proofs. Now, we introduce the notion of integrals for regular tree languages. But before, we give an elementary definition of integrals on trees.

Definition 5.2.2 *Given a special symbol $\lambda \notin \Sigma_n, n \geq 0$ and a symbol $f \in \Sigma_n$, the integral of the tree $\lambda(t_1, \ldots, t_n)$ is defined as follows:*

$$\int \lambda(t_1, \ldots, t_n)\, df = f(t_1, \ldots, t_n) \tag{5.6}$$

We generalize this definition to regular tree expressions.

Definition 5.2.3 *Let $\mathcal{N}$ be a set of n-tuple of regular tree expressions. The integral of $\mathcal{N}$ by $f \in \Sigma_n$ is defined as follows:*

$$\int \mathcal{N} df = \sum_{(E_1, \ldots, E_n) \in \mathcal{N}} f(E_1, \ldots, E_n)$$

Remark 5.2.4 *Notice that $\int \emptyset df = 0$ for $f \in \Sigma_{>0}$.*

Obviously, we should denote the tree language recognized by an integral.

Lemma 5.2.5

$$[\![\int \mathcal{N}\, df]\!] = \{ f(t_1, \ldots, t_n) \mid (E_1, \ldots, E_n) \in \mathcal{N}, t_1 \in [\![E_1]\!], \ldots, t_n \in [\![E_n]\!], f \in \Sigma_n \} \tag{5.7}$$

The following lemmas state some properties of integrals that can be easily verified.

Lemma 5.2.6 *Let $\mathcal{N}$ and $\mathcal{M}$ be two n-tuple regular tree expressions. Then we have:*

$$\int \mathcal{N} \cup \mathcal{M}\, df \equiv \int \mathcal{N}\, df + \int \mathcal{M}\, df$$

Proof. We have

$$\begin{aligned} \int \mathcal{N}\, df + \int \mathcal{M}\, df &= \sum_{(E_1, \ldots, E_n) \in \mathcal{N}} f(E_1, \ldots, E_n) + \sum_{(F_1, \ldots, F_n) \in \mathcal{M}} f(F_1, \ldots, F_n) \\ &= \sum_{(E_1, \ldots, E_n) \in (\mathcal{N} \cup M)} f(E_1, \ldots, E_n) \\ &= \int \mathcal{N} \cup \mathcal{M}\, df \end{aligned}$$

□

Lemma 5.2.7 *Let E be a regular tree expression. We have:*

$$\int \mathcal{N}._c E\ df = (\int \mathcal{N}\ df)._c E$$

Proof. We have

$$\begin{aligned}(\int \mathcal{N}\ df)._c E &= (\sum_{(E_1,\ldots,E_n)\in\mathcal{N}} f(E_1,\ldots,E_n))._c E \\ &= \sum_{(E_1,\ldots,E_n)\in\mathcal{N}} f(E_1,\ldots,E_n)._c E \\ &= \int \mathcal{N}._c E\ df\end{aligned}$$

□

We show now that integrals can be seen as the inverse operation of derivation. This can be proved by induction on the structure of the regular expression. But first we recall the principle of the partial derivation of a regular tree expression as follows [KM11]:

Definition 5.2.8 *Let $E, F, E_1, \ldots, E_n$ be regular tree expressions. The partial derivative of a regular expression is defined inductively as follows:*

$$\begin{aligned}\frac{\partial 0}{\partial a} &= \{0\}\ \textit{if}\ a \in \Sigma_0 \\ \frac{\partial f(E_1,\ldots,E_n)}{\partial g} &= \begin{cases}\{(E_1,\ldots,E_n)\}\ \textit{if}\ f = g \\ \{0\}\ \textit{otherwise}\end{cases} \\ \frac{\partial (E+F)}{\partial g} &= \frac{\partial E}{\partial g} \cup \frac{\partial F}{\partial g} \\ \frac{\partial E^{*c}}{\partial g} &= \frac{\partial E}{\partial g}._c E^{*c} \\ \frac{\partial E._c F}{\partial g} &= \begin{cases}\frac{\partial E}{\partial g}._c\{F\}\ \textit{if}\ c \notin E \\ \\ \frac{\partial E}{\partial g}._c\{F\} \cup \frac{\partial F}{\partial g}\ \textit{otherwise}\end{cases}\end{aligned}$$

Definition 5.2.9 *Let E be a regular tree expression. We define λ_E for a regular tree expression E as follows*

$$\lambda_E = \sum_{c\in([\![E]\!]\cap\Sigma_0)} c$$

That is to say:

$$\begin{aligned}
\lambda_c &= c \\
\lambda_{f(E_1,\ldots,E_n)} &= 0 \\
\lambda_{E_1+E_2} &= \lambda_{E_1} + \lambda_{E_2} \\
\lambda_{E_1 \cdot_c E_2} &= \lambda_{E_1} \cdot_c \lambda_{E_2} \\
\lambda_E^{*c} &= \lambda_c
\end{aligned}$$

Proposition 5.2.10 *Let E be a regular tree expression, then*

$$E \equiv \sum_{g \in \Sigma_>} \int g^{-1}E \, dg + \lambda_E \tag{5.8}$$

Proof. By induction over the structure of E, we have:

1. Case $E = a \in \Sigma_0$ then $\sum_{g\in\Sigma_>} \int g^{-1}E \, dg \equiv \sum 0 + \lambda_E \equiv a = E$.

2. Case $E = f(E_1,\ldots,E_n)$ then we have $\lambda_E = 0$. Thus:

$$\begin{aligned}
\sum_{g \in \Sigma_>} \int g^{-1}E \, dg &\equiv \int f^{-1}E \, df + \sum_{g \in \Sigma_>/f} \int g^{-1}E \, dg \text{ (Lemma 5.1.1(Equation 5.1))} \\
&\equiv \int f^{-1}E \, df + \sum 0 \\
&\equiv \int (E_1,\ldots,E_n) \, df = f(E_1,\ldots,E_n) = E
\end{aligned}$$

3. Case $E = E_1 + E_2$. We have $\lambda_E \equiv \lambda_{E_1} + \lambda_{E_2}$. Then:

$$\begin{aligned}
\sum_{g \in \Sigma_>} \int g^{-1}E \, dg + \lambda_E &= \sum_{g \in \Sigma_>} \int (g^{-1}E_1 \cup g^{-1}E_2) \, dg + \lambda_E \text{ (definition of derivative)} \\
&\equiv \sum_{g \in \Sigma_>} (\int g^{-1}E_1 \, dg + \int g^{-1}E_2 \, dg) + \lambda_E \text{ (Lemma 5.2.6)} \\
&\equiv \sum_{g \in \Sigma_>} \int g^{-1}E_1 \, dg + \sum_{g \in \Sigma_>} \int g^{-1}E_2 \, dg + \lambda_E \text{ (Identity element)} \\
&\equiv \sum_{g \in \Sigma_>} \int g^{-1}E_1 \, dg + \lambda_{E_1} + \sum_{g \in \Sigma_>} \int g^{-1}E_2 \, dg + \lambda_{E_2} \\
&\quad \text{(factorization of } \lambda_E \text{ and commutativity)} \\
&= E_1 + E_2 = E \text{ (induction hypothesis)}
\end{aligned}$$

4. Case $E = E_1^{*c}$ ($c \in \Sigma_0$) then we have two cases:

If $c \notin [\![E_1]\!]$ then $\lambda_E \equiv \lambda_{E_1} + c$. We have:

$$\begin{aligned}
E &\equiv E_1^{*c} + c \\
&\equiv E_1 ._c E_1^{*c} + c \text{ (Lemma 5.1.1)} \\
&\equiv (\sum_{g \in \Sigma_>} \int g^{-1}(E_1)\, dg + \lambda_{E_1}) ._c E_1^{*c} + c \text{ (induction hypothesis)} \\
&\equiv \sum_{g \in \Sigma_>} \int g^{-1}(E_1)\, dg ._c E_1^{*c} + \lambda_{E_1} ._c E_1^{*c} + c \text{ (right distribuvity)} \\
&\equiv \sum_{g \in \Sigma_>} \int g^{-1}(E_1) ._c E_1^{*c}\, dg + \lambda_{E_1} + c \text{ (Lemma 5.2.7, Lemma 5.1.1)} \\
&\equiv \sum_{g \in \Sigma_>} \int g^{-1} E\, dg + \lambda_E \text{ (definition of derivative)}
\end{aligned}$$

If $c \in [\![E_1]\!]$ then $\lambda_E \equiv \lambda_{E_1}$. Let F be a regular tree expression over Σ such that $F + c \equiv E_1$ and $c \notin [\![F]\!]$ then we have $\lambda_F \equiv \lambda_{E_1} + c$ and $E^{*c} \equiv (F + c)^{*c} \equiv F^{*c}$. Using the last result we get:

$$F^{*c} \equiv \sum_{g \in \Sigma_>} \int g^{-1} F^{*c}\, dg + \lambda_F + c$$

then

$$E_1^{*c} \equiv \sum_{g \in \Sigma_>} \int g^{-1} E_1^{*c}\, dg + \lambda_E$$

5. Case $E = E_1 ._c E_2$, then we distinguish two cases.

   If $c \notin [\![E_1]\!]$ then $\lambda_E \equiv \lambda_{E_1}$. We have:

$$\begin{aligned}
\sum_{g \in \Sigma_>} \int g^{-1}(E)\, dg + \lambda_E &= \sum_{g \in \Sigma_>} \int g^{-1}(E_1 ._c E_2)\, dg + \lambda_E \\
&\equiv \sum_{g \in \Sigma_>} \int g^{-1}(E_1) ._c E_2\, dg + \lambda_E \text{ (definition of derivative)} \\
&\equiv \sum_{g \in \Sigma_>} \int g^{-1}(E_1)\, dg ._c E_2 + \lambda_E \text{ (Lemma 5.2.7)} \\
&\equiv \sum_{g \in \Sigma_>} \int g^{-1}(E_1)\, dg ._c E_2 + \lambda_{E_1} ._c E_2 \text{ (Lemma 5.1.1)} \\
&\equiv (\sum_{g \in \Sigma_>} \int g^{-1}(E_1)\, dg + \lambda_{E_1}) ._c E_2 \text{ (right distributivity)} \\
&\equiv E_1 ._c E_2 = E \text{ (induction hypothesis)}
\end{aligned}$$

If $c \in [\![E_1]\!]$ then $\lambda_E \equiv \lambda_{E_1}.{}_c\lambda_{E_2} + \lambda_{E_2}$. We have:

$$\sum_{g\in\Sigma_>} \int g^{-1}(E)\, dg + \lambda_E = \sum_{g\in\Sigma_>} \int g^{-1}(E_1.{}_cE_2)\, dg + \lambda_E$$

$$\equiv \sum_{g\in\Sigma_>} \int g^{-1}(E_1).{}_cE_2 \cup g^{-1}(E_2)\, dg + \lambda_E$$

(definition of derivative)

$$\equiv \sum_{g\in\Sigma_>} \int g^{-1}(E_1).{}_cE_2\, dg + \sum_{g\in\Sigma_>} \int g^{-1}(E_2)\, dg + \lambda_E$$

(commutativity, Lemma 5.2.6)

$$\equiv \sum_{g\in\Sigma_>} \int g^{-1}(E_1)\, dg.{}_cE_2 + \sum_{g\in\Sigma_>} \int g^{-1}(E_2)\, dg + \lambda_{E_1}.{}_c\lambda_{E_2} + \lambda_{E_2}$$

(factorization of λ_E)

$$\equiv (\sum_{g\in\Sigma_>} \int g^{-1}(E_1)\, dg + \lambda_{E_1}).{}_cE_2 + \sum_{g\in\Sigma_>} \int g^{-1}(E_2)\, dg + \lambda_{E_2}$$

(Lemma 5.1.1)

$$\equiv E_1.{}_cE_2 + E_2 \equiv E_1.{}_cE_2 = E \text{ (induction hypothesis)}$$

□

Remark 5.2.11 *Integrals are defined only on arity and union operators. By analogy, we claim that it is sufficient to define derivations only on these two operations.*

Lemma 5.2.12 *Let $\mathcal{N}$ be an n-tuples of regular tree expressions. The derivation of a regular tree expression is a set of n-tuples that can be defined as follows:*

$$g^{-1}(f(E_1,\ldots,E_n)) = \begin{cases} \{(E_1,\ldots,E_n)\} \text{ if } f = g \\ \{0\} \text{ otherwise} \end{cases}$$

$$g^{-1}(E+F) = g^{-1}(E) \cup g^{-1}(F)$$

Therefore, derivation on c-product and c-closure can be proved using this result.

5.3 Synthesis of Regular Tree Expressions Using Integrals

The algorithm generating a regular tree expression from a tree automaton using integrals is based on associating, to each state in the automaton, a regular tree

expression denoting the regular language accepted by this state. Then, by integrating these regular expressions in a defined order combined with properties of integrals (combinatorial, substitution, etc.) along with regular expression properties, we deduce the regular expression denoting the whole automaton.

However, two problems have to be considered during this procedure. First, we should find a deterministic way to compute the corresponding regular tree expression. Then, we have to deal with the possible presence of cycles in the tree automaton.

The algorithm for the synthesis of a RTE from a TA is summarized as follows:

- Calculate the RTE of each state $q \in Q$.
- Resolve the cycles problem.
- Find a deterministic way to compute integrations.
- Use integrals and RTE properties to deduce the RTE of the whole automaton.

5.3.1 Regular Tree Expression for States

In the following, we consider the tree automaton $\mathcal{A} = (Q, \Sigma, Q_T, \Delta)$. Initially, for each state $q \in Q$, we denote by $L(q)$ the language accepted by the state q and $E(q)$ the regular tree expression such that $[\![E(q)]\!] = L(q)$.

Obviously:

$$L(\mathcal{A}) = [\![\sum_{q \in Q_T} E(q)]\!] \tag{5.9}$$

which is the regular tree expression that we want to deduce. We define $\lambda_{E(q)}$ for a state $q \in Q$ as follows.

$$\lambda_{E(q)} = \sum_{(c,q) \in \Delta} c \tag{5.10}$$

The following lemma defines a regular tree expression denoting a state $q \in Q$.

Lemma 5.3.1

$$E(q) \equiv \sum_{\substack{(f,q_1,\ldots,q_n,q) \in \Delta \\ n \geq 1}} \int \{(E(q_1), \ldots, E(q_n))\}\, df + \lambda_{E(q)}$$

Proof.

Let $(f, q_1, \ldots, q_n, q) \in \Delta$. We have $[\![E_{q_1}]\!] = L(q_1), \ldots, [\![E_{q_n}]\!] = L(q_n)$ (induction hypothesis).

Then

$$\int \{(E(q_1),\ldots,E(q_n))\}\ df = f(E(q_1),\ldots,E(q_n))$$

which denotes $f(L(q_1),\ldots,L(q_n))$.

Let $L'(q) = \bigcup_{(f,q_1,\ldots,q_n,q)\in\Delta} f(L(q_1),\ldots,L(q_n))$ and let $t = f(t_1,\ldots,t_n)$ be a tree in $T(\Sigma)$. We show that $t \in L(q) \Leftrightarrow t \in L'(q)$.

We have by definition: $t \in L(q) \Leftrightarrow q \in \delta(t)$.

Then:

$$\begin{aligned} q \in \delta(t) &\Leftrightarrow \exists(f,q_1,\ldots,q_n,q) \in \Delta, (\forall 1 \leq i \leq n, q_i \in \delta(t_i)) \\ &\Leftrightarrow \exists(f,q_1,\ldots,q_n,q) \in \Delta, (\forall 1 \leq i \leq n, t_i \in L(q_i)) \\ &\Leftrightarrow \exists(f,q_1,\ldots,q_n,q) \in \Delta, t \in f(L(q_1),\ldots,L(q_n)) \\ &\Leftrightarrow t \in L'(q) \end{aligned}$$

□

5.3.2 Dealing With Cycles in Regular Expression Synthesis

We distinguish two types of cycles: direct and indirect cycles. Direct cycles occur with transitions of the form: $(f,q_1,\ldots,q_n,q), \exists q_i \in \{q_1,\ldots,q_n\} | q_i = q$.

To solve this problem, we define for each state the two following regular expressions. We denote by $\mathcal{F}(q)$ regular expressions without cycles and $C(q)$ the ones with cycles.

$$\begin{aligned} \mathcal{F}(q) &= \sum_{(f,q_1,\ldots,q_n,q)\in\Delta} \int \{(E(q_1),\ldots,E(q_n))\}\ df + \lambda_{E(q)} \text{ such that } \forall q_i \in q_1,\ldots,q_n : q_i \neq q \\ C(q) &= \sum_{(f,q_1,\ldots,q_n,q)\in\Delta} \int \{(E(q_1),\ldots,E(q_n))\}\ df \text{ such that } \exists q_i \in q_1,\ldots,q_n : q_i = q \end{aligned}$$

Let $x \notin \Sigma_0$ be an extra leaf symbol and let $p,q \in Q$. We define $\alpha_x(p,q)$ as follows.

$$\alpha_x(p,q) = \begin{cases} x \text{ if } p = q \\ E(p) \text{ otherwise} \end{cases}$$

$C_x(q)$ is the substitution of every occurrence of $E(q)$ by the symbol x in $C(q)$.

$$C_x(q) = \sum_{(f,q_1,\ldots,q_n,q)\in\Delta} \int \{(\alpha_x(q_1,q),\ldots,\alpha_x(q_n,q))\}\ df \text{ such that } \exists q_i \in q_1,\ldots,q_n | q_i = q$$

Obviously:

Lemma 5.3.2

$$C(q) = C_x(q).{}_x E(q)$$

Direct cycles can be solved using the following lemma.

Lemma 5.3.3 *Let $q \in Q$. If $C(q) \neq 0$ then:*

$$E(q) = (C_x(q))^{*x}.{}_x \mathcal{F}(q) \tag{5.11}$$

P r o o f . First, we have:

$$\begin{aligned} E(q) &= C(q) + \mathcal{F}(q) \\ &= C_x(q).{}_x E(q) + \mathcal{F}(q) \text{ (Lemma 5.3.1)} \end{aligned}$$

We replace $E(q)$ by $(C_x(q))^{*x}.{}_x \mathcal{F}(q)$ in the right side:

$$\begin{aligned} C_x(q).{}_x E(q) + \mathcal{F}(q) &\equiv C_x(q).{}_x((C_x(q))^{*x}.{}_x \mathcal{F}(q)) + \mathcal{F}(q) \\ &\equiv C_x(q).{}_x((C_x(q))^{*x}.{}_x \mathcal{F}(q)) + x.{}_x \mathcal{F}(q) \\ &\equiv (C_x(q).{}_x(C_x(q))^{*x} + x).{}_x \mathcal{F}(q) \\ &\equiv (C_x(q))^{*x}.{}_x \mathcal{F}(q) \end{aligned}$$

□

Direct cycles resolution can be used when dealing with indirect cycles too. In fact, a infinite tree generation can occur even if no direct cycles exist.

Example 5.3.4 *Let $\mathcal{A} = (\{p,q\},\{(f,1),(g,a),(a,0)\},\{p\},\{(f,p,q),(g,q,p),(a,p),(a,q)\})$ be a tree automaton. The transitions (f,p,q) and (g,q,p) constitute an indirect cycle.*

We have

$$E(p) = \int \{(E(q))\}\, dg + a = g(E(q)) + a$$
$$E(q) = \int \{(E(p))\}\, df + a = f(E(p)) + a$$

$E(p)$ can be seen as a cycled regular tree expression if we substitute $E(q)$ by its formula :

$$E(p) \equiv g(f(E(p)) + a) + a$$

Thus, Lemma 5.3.3 can be directly used to find the appropriate regular tree expression.

The other problem that we have to deal with is which state should we use first in the cycle computation process. In other words, should we compute $E(p)$ and use it next for $E(q)$ computation or the inverse. Consequently, there are numerous ways to compute a regular tree expression of a cyclic tree automaton. Therefore, we propose a deterministic order to make the construction unique.

For this reason, we define what we call *state level*. It consists of the minimum path between a state and a final state. Clearly, a level of a final state is equal to 1.

Let $p,q \in Q$ be two states, then p is a *direct descendant* of q if there exists a transition $(f,q_1,\ldots,p,\ldots,q_n,q), f \in \Sigma, q_1,\ldots,q_n \in Q$. The sequence $q_1,\ldots,q_n$ is a *path* if $\forall i \in 1,\ldots,n-1 : q_{i+1}$ is a direct descendant of q_i. (such that $q_i \in Q$). The length of a path $s = 1,\ldots,n$ noted $\|s\|$ is n. Here, a *state level* is:

Definition 5.3.5 *Let $q \in Q$. Let $\hat{q}$ be the set of all finite paths $q_1,\ldots,q_n,q$ such that $q_1 \in Q_T, q_2,\ldots,q_n \in Q$, $n \leq |Q|$. The level of q noted $Lev(q)$ is defined as follows.*

$$Lev(q) = \min_{s \in \hat{q}} \|s\| \tag{5.12}$$

Corollary 5.3.6 *Let $q \in Q_T$ then $Lev(q) = 1$.*

Besides that, detecting indirect cycles is very important for the synthesis of a regular tree expression. Therefore, for each cycle, we define an order on states included in one cycle.

A cycle is the sequence $p,q_1,\ldots,q_n,p$ such that for all $q_i,q_{i+1},i \in 1,\ldots,n-1$ there exists a transition in which q_i occurs and q_{i+1} is its output. Thus, we consider the binary relation $\preceq \subseteq Q^2$ defined as follows.

Definition 5.3.7 *Let p,q be two states, then*

$$(p \preceq q) \Leftrightarrow (Lev(p) \leq Lev(q)) \wedge (p \text{ and } q \text{ belong to the same cycle})$$

Algorithm 1 gives for each state, the level and the set of cycles in which it figures. The following notations are used by the algorithm.

- $D(q)$ is the set of states participating in a transition whose target is q, $D(q) = \{q', \delta(f,q_1,\ldots,q',\ldots,q_n) = q\}$,
- $P(q)$ is the set of states from q to final states. It is used cycles computation. This list is initially empty.

Recall that all states levels are initialized to zero. *temp* is a temporary set including states to be computed in one iteration. *tempnxt* collects states needed in the next iteration. *CurrentLevel* computes the state levels and it is set to zero and incremented at each iteration. The algorithm ends after $|Q|$ iterations.

Algorithm 1: Function *CyclesLevelComp*

```
1  CurrentLevel ← 1;
2  temp,tempnxt,← ∅;
3  foreach q ∈ Q_T do
4      Lev(q) = 0;
5  temp ← Q_T;
6  repeat
7      foreach q ∈ temp do
8          foreach q′ ∈ D(q) do
9              tempnxt ← tempnxt ∪ {q′};
10             if q′ ∈ P(q) then
11                 q is cyclic;
12                 P(q) ← P(q) ∪ {q′};
13             if lev(q′) = 0 then
14                 Lev(q′) ← CurrentLevel;
15     temp ← tempnxt;
16     tempnxt ← ∅;
17     CurrentLevel ← CurrentLevel + 1;
18 until CurrentLevel = |Q|;
```

According to the order relation, we distinguish three types of states:

- $Free$ is the set of states for which we can deduce the RTE denoting their language directly as we have $p \in Free \Leftrightarrow P(q) = \emptyset$.
- $Cycled$ is the set of cyclic states.
- $Linked = Q - (Free \cap Cycled)$ is the set containing states that need other rational expressions computations but do not participate in any cycle. $q \in Linked \Leftrightarrow (q$ is not cyclic and $P(q) \neq \emptyset)$.

Obviously, initially $Free$ contains all the states of the tree automaton.

In order to synthesis regular tree expression, we should transform indirect cycles into direct ones. This is done by substitutions on cyclic regular tree expressions w.r.t the defined order. Consequently, no state's RTE is substituted in another state's RTE with higher level.

Let $q \in Q$ be a cycled state without direct cycle ($E(q) = \mathcal{F}(q)$). Here for all cyclic state q' figuring in the path set $P(q)$, we substitute $E(q')$ by its computation. If $E(q)$ become with direct cycle, we use then Lemma 5.3.3. Otherwise we do the same substitution for all states q'.

The regular tree expression constructed by Algorithm 2 denotes the language accepted by the tree automata given as input. This algorithm uses:

- The function $ComputeRTE$ which returns $E(q)$ for a state q according to Lemma 5.3.1 and stores resulting regular tree expressions in $M(q)$. For every state q, we associate a set of predecessors $Pred(q) = \{q', \exists(f, q_1, \ldots, q, \ldots, q_n, q') \in \Delta\}$ for $f \in \Sigma_n$ and $q_1, \ldots, q_n \in Q$.

- The function $Substitue(E, s)$ that substitutes the regular tree expression E in all $M(q), q \in s$. Thus, for every state q from $Free$, $E(q)$ is substituted in all states from $Pred(q)$. For cyclic states, we order them according to their levels and the computation starts with the state of the higher level and so on.

- The function $ComputeCycle$ which outputs the regular tree expression of a state q if it contains a direct cycle (according to Lemma 5.3.3). If the cyclic state figures in a linked or a free state's path list, then the regular tree expression of the last one is substituted in the regular tree expression of the state in question. By this way, all indirect cycles are transformed into direct ones.

At the end of the algorithm, the regular tree expression of the automaton is the sum of all regular tree expressions in M that the corresponding states figure in the final states list.

Algorithm 2: Algorithm *ComputeRTEfromFTA*

Input: $\mathcal{A} = (\Sigma, Q, Q_T, \Delta)$

Output: E: Regular Tree Expression

```
1  Computed ← ∅;
2  foreach q ∈ Q do
3      M(q) ← ComputeRTE(q);
4  foreach q ∈ Free do
5      Computed ← Computed ∪ {q};
6      Substitue(M(q), Pred(q));
7  foreach q ∈ φ(Cycled) do
       /* φ orders Cycled states according to their levels        */
8      if M(q) contains direct cycles then
9          M(q) ← ComputeCycle(q);
10         Substitue(M(q), Pred(q));
11 repeat
12     foreach q ∈ Q − Computed do
13         if No regular expression index figures in M(q) then
14             Computed ← Computed ∪ {q};
15         else
16             foreach q' ∈ Pred(q) do
17                 Substitute(M(q'), {q});
18 until Computed = Q;
19 return ∑_{q∈Q_T} M(q);
```

Theorem 5.3.8 *The algorithm of synthesis of a RTE from a TA using integrals requires* $O(|Q|^2.R)$ *time complexity.*

Q is the set of states and R is the maximal rank of the alphabet Σ.

Proof. On the one hand, in the function calculating cycles 1 for each state $q \in Q$, the *repeat* loop (line 6) requires $O(|Q|)$, the inner *Foreach* loop (line 8) requires $O(R)$ and the other *foreach* loop (line 7) requires $O(|Q_T|)$ which we bound by $O(|Q|)$. Then the whole function requires $O(|Q|^2.R)$ time complexity.

On the other hand, the algorithm computing the RTE from TA 2 requires $O(|Q|)$ for the three first *foreach* loops (lines 2, 4 and 7), and $O(|Q|)$ for the *repeat* loop (line 11). Then, the synthesis of a RTE from a TA requires $O(|Q|^2.R) + 2.|Q|$ which is equal to $O(|Q|^2.R)$. □

Example 5.3.9 *We consider the tree automaton in Figure 5.1.*

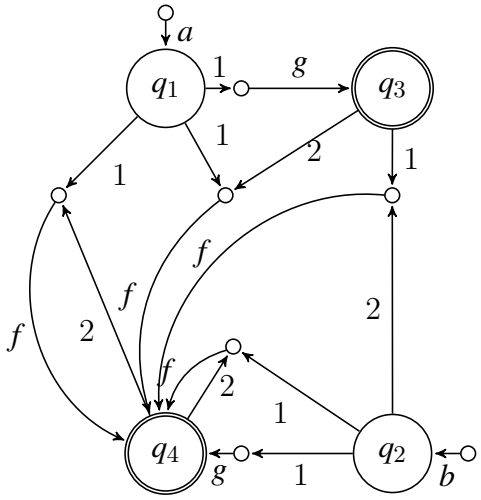


Figure 5.1: Example of a tree automaton

After cycles and levels computation, we get the following table.

	Free	*Cycled*	*Linked*	*Lev*
q_1	•			2
q_2	•			2
q_3			•	2
q_4		•		*1*

Here, only one cycled state exists. There is no need to use the levels ordering. M *computation gives:*

$$M(q_1) = \int \lambda\ da = a$$

$$M(q_2) = \int \lambda\ db = b$$

$$M(q_3) = \int \{(E(q_1))\}\ dg = g(E(q_1))$$

$$M(q_4) = \int \{(E(q_1),E(q_4)),((E(q_1),E(q_3)),((E(q_3),E(q_2)),((E(q_2),E(q_4))\}\ df$$

$$= f(E(q_1),E(q_4)) + f(E(q_1),E(q_3)) + f(E(q_3),E(q_2)) + f(E(q_2),E(q_4))$$

After substituting $E(q_1), E(q_2)$ *and* $E(q_3)$ *in their predecessors. We resolve the direct cycle present in* $E(4)$*. Thus:*

$$\begin{aligned}\mathcal{F}(q_4) &= f(a, g(a)) + f(g(a), b)\\ C(q_4) &= f(a, E(q_4)) + f(b, E(q_4))\\ M(q_4) &= (f(a,x) + f(b,x))^{*x}.{}_x f(a, g(a)) + f(g(a), b)\end{aligned}$$

As q_4 *is the unique final state, then*

$$E_{\mathcal{A}} = (f(a,x) + f(b,x))^{*x}.{}_x f(a, g(a)) + f(g(a), b)$$

5.4 Conclusion

In this chapter we have presented our generalization of integration from regular string expressions to trees. We have given the principles and properties of integrals in trees. We have also shown how integrals are considered as the inverse operation of tree derivatives. This generalization is useful for synthesizing a regular tree expression from a given tree automaton.

Chapter 6

Tree Pattern Matching

Tree pattern matching algorithms play an important role in many applications such as compilers, validation of XML documents, automatic proofs, etc. This problem can be defined as the search for all occurrences of one or more given trees (patterns) in a subject tree. There exist two types of tree pattern matching: subtree and tree template matching. While subtrees consist of only exact and fixed-labeled nodes, tree templates have some of their leaves variables which match any subtree [Flo12].

In this chapter, we present two new approaches to solve the tree pattern matching problem: the first one using our generalization of Thompson tree automaton (Chapter 4) and the other one deals rather with fixed tree pattern matching, that is the research of a single tree pattern in a target subject tree. This algorithm has been inspired by the factor and suffixes oracles in string pattern matching. But first, we start with a brief literature overview of the existing tree pattern matching algorithms in trees.

6.1 Tree Pattern Matching: Literature Overview

A tree pattern (for sake of simplicity we use pattern) is a tree in which variable leaves may exist, that is leaves with a symbol which can be substituted by any other subtree of the tree language. This symbol is referred to as υ. Tree pattern matching problem is the identification of occurrences of one or more tree patterns in a subject tree.

Definition 6.1.1 *For each tree $t \in T_\Sigma$ and each pattern $p \in T_{\Sigma\cup\{\upsilon\}}$, a subtree in t matches the pattern p in a node n means that $p = t/n$, where t/n is the subtree of t rooted by n.*

The input of a tree pattern matching algorithm, i.e. the pattern, might be a simple tree, a set of finite trees or even a set of tree language represented by a regular tree expression. Its output is the nodes of the subject tree that match the pattern.

Tree pattern matching can be considered as an extension of string pattern matching. Several algorithms have appeared in the literature [Cle08, FIJ+12, HO82, IWIU12, LW00, MS98, RUG97].

While most tree pattern matching literature appeared from the early 1980s onwards, the 1975 PhD thesis of Kron was an earlier exception [Kro75]. Kron uses so-called orthogonal tree automata, automata that read the sequence of states assigned to the sequence of children of a node to determine the state to be assigned to the parent node.

In 1982, Hoffmann and O'Donnell in [HO82] presented a more extensive study of tree pattern matching. They proposed several algorithms solving the tree pattern matching problem for both bottom-up and top-down approaches. First, the bottom-up method generalizes string matching. The idea here is to find at each node in the subject tree, all patterns or parts of patterns matching this node. Interestingly, Hoffmann and O'Donnell do not mention bottom-up tree automata, even though their bottom-up method in essence corresponds to tabulating such an automaton. Their top-down approach reduces tree matching to a string-matching problem by using a string-path matching automaton.

Another solution to the tree pattern matching problem exists. The principle is to encode the subject tree as strings, allowing the use of string pattern matching techniques. In [LW00] a generalization of the Knuth-Morris-Pratt algorithm of string pattern matching into trees is given. In the Knuth-Morris-Pratt algorithm the method used is the precomputation of shifts [RUG97]. A more recent work using linearization is presented in [TJMC15], where a backward tree pattern matching technique is approached that uses ideas from Boyer-Moore and Horspool's string pattern matching algorithms.

Other techniques use a pushdown automaton [MS98, FIJ+12]. In [FIJ+12] the method used is analogous to the construction of string pattern matchers: for given patterns, a nondeterministic pushdown automaton is created which is then determinized for efficiency reasons.
In a different way, Itokawa et al [IWIU12], use a depth-first unary degree sequence (DFUDS), which is a data structure associated with an ordered tree and expressing the features of a graph structure. Then, they have proposed a pattern matching algorithm that uses the DFUDS data structure to determine whether or not a given tree has features of a tree pattern.

An extensive study about tree pattern matching can be found in [Bel14] where a survey on these algorithms has been conducted along with a comparison between the different algorithms and their complexity.

6.2 Motivation and Aims

All the previously discussed tree pattern matching algorithms consider the pattern as a tree or a *finite* set of trees. In our work, we focus on a more generalized problem by using patterns represented by a regular tree expression.

A basic and well-known algorithm for regular expression pattern matching for

strings is Thompson algorithm [Tho68]. The principle of this algorithm consists of building, by induction, an automaton from the regular expression of the pattern and in a second step to process the subject text using the constructed automaton in order to identify the different pattern occurrences. This algorithm has been used in many practical tools such as the *grep* utility on Linux [Goe95, AA04].

We propose hereafter a tree pattern matching algorithm inspired by Thompson pattern matching for strings. This algorithm is based on the construction of a special tree automaton that can be viewed as a generalization of Thompson one for strings. Our proposal might be useful in all fields that need a lookup for one or multiple patterns in a subject tree, especially when these patterns are represented by a regular tree expression. For example: instruction selection in automatic code generation [AG85, FHP92], genetics [Gie98, DDP+05], term rewriting [HO82, Grä91], verification of network protocol and cryptography [GL00, GK00]. These results have been published in [BCCZ18].

For our Thompson tree automaton-based pattern matching algorithm, we want to index the nodes of the subject tree. To do so, we mark each node as presented in the following definition.

Definition 6.2.1 *Let $f \in \Sigma$. Marking the nodes of a tree t is defined inductively as follows.*

$$\text{Mark}(f) = \begin{cases} f_1 & \text{if } f \text{ is the root symbol} \\ f_{\text{Mark}(Parent(f))\text{pos}(f)} & \text{otherwise} \end{cases}$$

where $\text{pos}(f)$ *is the position of a node* f *among its sibling.*

Example 6.2.2 *For example, let $t = f(f(a,b),h(g(d)))$ be a tree. After marking t , we get the following indexed tree: $f_1(f_{11}(a_{111},b_{112}),h_{12}(g_{121}(d_{1211})))$.*

Definition 6.2.3 *Let Σ_M be the set of marked symbols of Σ. We define the mapping h from Σ_M to Σ, which for a marked symbol $f_{u_1 \ldots u_k}$ gives its corresponding symbol in Σ, that is f.*

6.3 Thompson Tree Pattern Matching Algorithm

By analogy to words, we will use the extended Thompson automaton in tree pattern matching. Let E be a regular tree expression representing the pattern and t a marked and linearized tree.

So, looking for occurrences of a regular tree expression E in a subject tree t is accomplished in two steps:

- Constructing Thompson tree automaton for E,
- Running the constructed automaton on t: each time the final state is reached, an occurrence of the pattern E has been recognized.

The construction proposed in the third chapter allows us to perform two kinds of pattern matching:

1. Top-down pattern matching by constructing a Thompson tree automaton for the regular tree expression $E \cdot_{\upsilon} (\Sigma \cup \{\upsilon\})^{*,\upsilon}$, where E is the regular expression of the tree pattern and $\upsilon \in \Sigma_0$ represents the variable leaf.
2. Bottom-up pattern matching by using the Thompson tree automaton of the regular tree expression E. We don't need to add a variable leaf as we perform the pattern matching from leaves and if a pattern exists it must be a subtree of the subject tree.

Note that both the top-down and the bottom-up versions of Thompson tree automaton are nondeterministic, but the top-down one is highly nondeterministic compared to the bottom-up one, thus the pattern matching takes more time. Furthermore, in general the bottom-up automaton could be determinised while the top-down one could not be, given the limited power of deterministic top-down tree automata [MNS08]. For this reason we will develop hereafter only the bottom-up approach.

6.3.1 Tree Pattern Matching Algorithm

We will not make the constructed automaton deterministic, in order to keep the reduced number of states and transitions. In fact, to verify that a tree t is recognized by a tree automaton Th_E, we will simulate a determinization by activating in each step of the traversal of the constructed automaton, states that are reachable from initial states.

Before giving the tree pattern matching algorithm, we define two functions *Skip_Epsilon* and *Move*.

Definition 6.3.1 *For a state $q \in Q$, we define the set $Q_q^{\varepsilon} = \{p \mid \varepsilon(q) \longrightarrow p \in \Delta\}$.*

This definition can be extended to set of states.

Definition 6.3.2 *For a subset $P \subseteq Q$, we define the set $Q_P^\varepsilon = \bigcup_{q \in P} Q_q^\varepsilon$.*

These two definitions are implemented in Algorithm 3 (*Skip_Epsilon*) that allows the algorithm to get over the ε-transitions. Let $P \subseteq Q$ be a set of states. We define the set $\lambda(P) = \{q \in P \mid Q_q^\varepsilon = \emptyset\}$.

Algorithm 3: Function *Skip_Epsilon*

Input: P : Set of states.

Output: Set of reachable states from P after skipping ε-transitions.

```
1 R ← P ;
2 X ← ∅;
3 while R ≠ ∅ do
4 |   X ← X ∪ λ(R);
5 |   R ← Q_R^ε;
6 return X ;
```

In function *Skip_Epsilon* (Algorithm 3), we calculate the set of reachable states from a set of states P. In every iteration of the loop *while* (line 3), we add the set $\lambda(R)$, which represents the subset of states of P that don't lead to any other state by an ε-transition, to the output set (line 4). The set R, initialized by P, contains at each step the reachable states after skipping one ε-transition (line 5). These instructions are repeated until there is no more ε-transitions to be reached.

According to the inductive construction of Thompson tree automata and using Definition 6.3.2, we can deduce the following property which guarantees that in a Thompson tree automaton, no cycles of ε-transitions exist.

Property 6.3.3 *For a subset $P \subset Q$, $Q_P^\varepsilon \neq P$*

Corollary 6.3.4 *The function* Skip_Epsilon *has $\mathcal{O}(|E|)$ time complexity.*

Proof. From Property 6.3.3 we deduce that the loop *while* has a finite number of iterations. Let M be this number of iterations and R_i be the subset of states calculated in the i$^{\text{th}}$ iteration. So, this loop is calculated in $\sum_{i=1}^{M} |\lambda(R_i)|$. Using Property 4.6.1, $\sum_{i=1}^{M} |\lambda(R_i)| \leqslant 2|Q|$. Therefore, the function *Skip_Epsilon* has $\mathcal{O}(|Q|)$ time complexity. Using Corollary 4.6.2, this complexity can be bounded by $\mathcal{O}(|E|)$. □

Definition 6.3.5 *Let Q_f be the set of reachable states after reading f, that is* $Q_f = \{p \in Q \mid \exists\, q_1, \cdots, q_n \in Q, f(q_1, \cdots, q_n) \longrightarrow p \in \Delta\}$.

Definition 6.3.6 *Let Q_f^i be the set of states that are the i^{th} child of f, that is* $Q_f^i = \{q_i \in Q \mid f(q_1, \cdots, q_i, \cdots, q_n) \longrightarrow p \in \Delta\}$.

These two definitions are used in Algorithm 4 (function *Move*) in order to define the set of reachable states after reading a symbol $f \in \Sigma_n$.

In the function *Move* (Algorithm 4) we start by computing sets Q_f and Q_f^i for $i = 1 \ldots n$, where n is the arity of the symbol f. Then, we select from the set Q_f states for which all children already exist in the underlying set Q_f^i. Therefore, we get the output set of states reachable by reading the symbol f.

Algorithm 4: Function *Move*

Input: $P_1, P_2, \ldots, P_n$: Sets of states. $f \in \Sigma_n$.

Output: R : Set of states.

```
1 Compute Q_f and Q_f^i for i = 1,...,n
2 R ← ∅;
3 foreach (p ∈ Q_f) do
4     //Let q_1,...,q_i,...,q_n ∈ Q such that f(q_1,...,q_i,...,q_n) ⟶ p ∈ Δ
5     if (q_i ∈ (P_i ∩ Q_f^i), for i = 1,...,n) then
6         R ← R ∪ {p};
7 return R;
```

Corollary 6.3.7 *The function* Move *has $O(|E|)$ time complexity.*

Proof. Both the **foreach** loop (line 3) and the membership test (line 5) require $O(r|Q|)$ time complexity, where r is the maximal arity for Σ, that is an $O(|Q|)$ time complexity. As we want to express the complexity in terms of the input regular tree expression, we use Corollary 4.6.2 to bound $|Q|$ by $|E|$. So Algorithm 4 requires $O(|E|)$ time complexity. □

Tree pattern matching algorithm using Thompson tree automaton is presented in Algorithm 5. This algorithm takes as input a *marked* subject tree t and the Thompson tree automaton Th_E of the pattern's regular tree expression E. We run the automaton using the functions *Skip_Epsilon* (Algorithm 3) and *Move*(Algorithm 4) in order to find occurrences of the pattern in the subject tree. Each time the final

state q^E is reached, an occurrence of the pattern is found, and the symbol leading to the final state is added to the output set.

Algorithm 5: Algorithm TPM

Input: $t_{u_1 \ldots u_k}$: a node of the marked subject tree.

Th_E : Thompson automaton for E.

Output: $P_{u_1 \ldots u_k}$: set of states.

```
1 if (ar(t) = 0) then
2 |   P_{u1...uk} ← Skip_Epsilon(Move(h(t_{u1...uk})));
3 else
4 |   P_{u1...uk} ←
  |   Skip_Epsilon(Move(TPM(t1),TPM(t2),...,TPM(tn),h(t_{u1...uk})));
5 |   // n = ar(t)
6 if (q^E ∈ P_{u1...uk}) then
7 |   occ ← occ ∪ {t_{u1...uk}};
8 |   P_{u1...uk} ← P_{u1...uk} \ {q^E};
9 return P_{u1...uk};
```

In the TPM algorithm (Algorithm 5) we associate a set $P_{u_1 \ldots u_k}$ to each node $t_{u_1 \ldots u_k}$ of the subject tree t. As the algorithm goes on these sets maintain the reachable states during the traversal of the automaton by reading nodes of t. For each node $t_{u_1 \ldots u_k}$, we call the functions *Move* and *Skip_Epsilon* in order to calculate the new set $P_{u_1 \ldots u_k}$ taking as input parameters the symbol $t_{u_1 \ldots u_k}$ and the sets calculated recursively $P_{u_1 \ldots u_k}$. If the final state of the pattern's Thompson tree automaton is included in the set $P_{u_1 \ldots u_k}$, we add the symbol $t_{u_1 \ldots u_k}$ to the set of nodes matching the pattern occ.

Theorem 6.3.8 *The tree pattern matching algorithm using Thompson tree automaton requires $O(|t||E|)$ time complexity.*

Proof. In Algorithm 5, the recursive call of TPM allows the process of all nodes in t, that is $|t|$. Using the functions *Move* and *Skip_Epsilon* for each node, we get an $O(|t||E|)$ time complexity. □

6.3.2 Example of Tree Pattern Matching Using Thompson Automaton

Let us consider the regular tree expression of Example 4.4 $E = (f(a,b) + g(c) \cdot_c d)^{*,d}$, and $t = a_{111}b_{112}d_{1211}f_{11}g_{121}h_{12}f_1$ a *marked* and *linearized* subject tree. Figure 6.1 recalls Thompson tree automaton Th_E constructed for E.

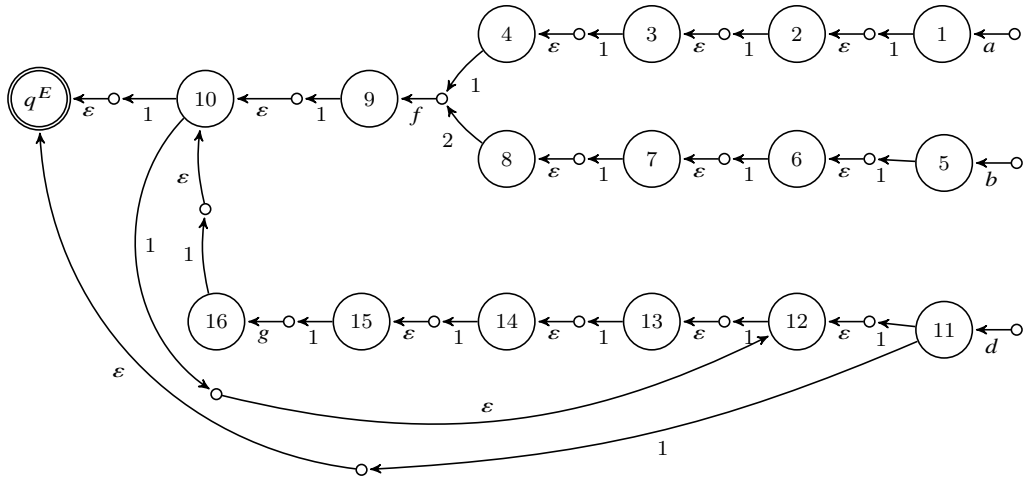
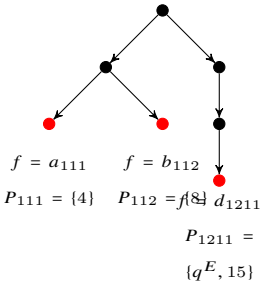
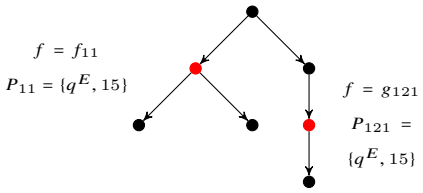


Figure 6.1: Thompson tree automaton of the pattern

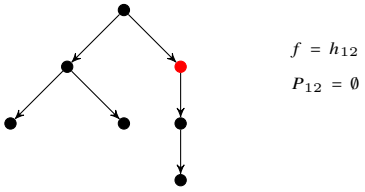
In order to determine nodes that match the pattern's regular tree expression, we run Algorithm 5 with t and Th_E as input parameters. Figure 6.2 shows the successive construction of sets $P_{u_1 \ldots u_k}$ by using the functions *Move* and *Skip_Epsilon*.



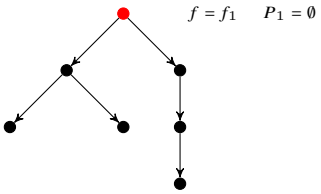
(a) Step 1: a match in d_{1211}



(b) Step 2: two matches in f_{11} and g_{121}



(c) Step 3: no matches



(d) Step 4: no matches

Figure 6.2: Example of running TPM algorithm using Thompson tree automaton

6.4 Tree Pattern Matching Using Tree Suffixes Automaton

Fixed string pattern matching is defined as the search for one or multiple occurrences of a specific fixed string in a subject text. This field has been quite well investigated in the literature [MM01, KMP77, BM77, Hor80, AH74, CCG+94, CW79, NR98, CMM99].

By the late nineteens, new structures (viz. suffix oracle and factor oracle), which are special finite automata, have been developed in order to optimize the string matching process [ACR99, CMM99, CMM01, MM05]. These new structures have been used in different applications [AD04, KW05, LL02a, LL02b, LLA03].

6.4.1 Factor Oracles

A factor oracle is a data structure for weak factor recognition. It can be described as an automaton built on a string p of length m that is:

- acyclic,
- recognizes at least all factors of p,
- has $m + 1$ states which are all final,
- and has m to $2m - 1$ transitions.

Factor oracles have been introduced in [ACR99] as an alternative to exact factor recognition in on-line string pattern matching algorithms. In such algorithms, a window on a text is read backward while attempting to match a factor. If the match fails, the window is shifted to the mismatching character.

Instead of an automaton recognizing exactly the set of factors of a string, it is possible to use a factor oracle though it recognizes more strings than just the factors. The advantage of using factor oracles is that they are easier to construct and take less space compared to suffix, factor and subsequence automata [CZW03].

6.4.2 Generalization of Suffixes Automata to Trees

We generalize the idea of fixed string matching to trees. That is to say, we propose a new method similar to the suffix and factor oracles in strings to trees. First, we construct a bottom-up tree automaton recognizing all the suffixes of a tree pattern.

Then, we generalize the matching algorithm using the sliding windows mechanism in strings to trees.

In the following, suffixes automaton in trees is defined along with its validity proofs. Then, we propose a tree pattern matching algorithm using the constructed automaton.

Definition 6.4.1 *Let $t(t_1,\ldots,t_n) \in \Sigma_T$ be a tree. We denote by $Suff(t)$ the set of suffixes of t defined by $Suff(t) = \{s \in \Sigma_T | \exists n \in \Sigma, t/n = s\}$.*

Note that t/n denotes the subtree of t rooted at n. In other words, $Suff(t)$ is the set of all subtrees of the tree t.

In this algorithm we need the *Height* of trees defined in 1.3.7. Recall that the *Height* of a tree (subtree) t is the *Height* of its root.

6.4.3 Tree Suffixes Automata

We propose to construct a bottom up tree automaton $\mathcal{A}$ that recognizes the set of all suffixes of a tree t, i.e. $Suff(t)$. In this construction, we also need the indexed representation of tree defined for the previous tree pattern matching algorithm using Thompson's automata. We propose Algorithm 6 for indexing trees where $pos(n)$ represents the position of the node n among its siblings.

Algorithm 6: Function *Index_tree*

Input: t: tree

Output: $Mark(t)$: indexed tree.

```
1 Mark(t) ← t;
2 foreach (n node of Mark(t)) do
3     if (n = root(t)) then
4         n ← n1 ;
5     else
6         n ← nIndex_tree(parent(n))pos(n);
7 return Mark(t) ;
```

Let $\mathcal{A}_P = (Q_t, \overline{\Sigma_t}, Q_T, \Delta_t)$ be the tree suffixes automaton of t where Q_t is the set of states, $\overline{\Sigma_t}$ is the marked alphabet Σ, Q_T is the set of final states, and Δ_t is the transition function. We give the following steps for $\mathcal{A}_t$ construction:

- For each node $a_{u_1,\ldots,u_n} \in Mark(t)$ and $a \in \Sigma_0$, add a state labeled by $q_{u_1,\ldots,u_n}$ to Q_t and a transition $h(a_{u_1,\ldots,u_n}) \longrightarrow q_{u_1,\ldots,u_n}$ to Δ_t.
- For each node $f_{u_1,\ldots,u_n} \in Mark(t)$ and $f \in \Sigma_m$, add a state labeled by $q_{u_1,\ldots,u_n}$ to Q_t and a transition $h(f_{u_1,\ldots,u_n})(q_{u_1,\ldots,u_n 1} \ldots, q_{u_1,\ldots,u_n m}) \longrightarrow q_{u_1,\ldots,u_n}$ to Δ_t.
- Make all the states accepting ones: $Q_T = Q_t$.

Algorithm 7 presents the tree suffixes automaton construction.

Algorithm 7: Function *ConstructSuffixTreeAutomaton*

Input: t: indexed tree

Output: $\mathcal{A}_t$: tree suffixes automaton.

```
1  A_t.Q_t ← ∅;
2  A_t.Δ_t ← ∅;
3  foreach (n_{u_1,...,u_n} ∈ t) do
4  |   A_t.Q_t ← A_t.Q_t ∪ q_{u_1,...,u_n};
5  |   if (ar(n_{u_1,...,u_n}) = 0) then
6  |   |   A_t.Δ_t ← A_t.Δ_t ∪ {h(n_{u_1,...,u_n}) ⟶ q_{u_1,...,u_n}};
7  |   else
8  |   |   A_t.Δ_t ← A_t.Δ_t ∪ {h(n_{u_1,...,u_n})(q_{u_1,...,u_n 1} ..., q_{u_1,...,u_n ar(n)}) ⟶ q_{u_1,...,u_n}};
9  A_t.Q_T ← A_t.Q_t;
10 return A_t ;
```

We deduce the following property for tree suffixes automata.

Property 6.4.2 *The size of a tree suffixes automaton $\mathcal{A}_t$ is $|t|$.*

Example 6.4.3 *Let $\Sigma = \{f(2), b(0), c(0)\}$ and a tree $t = f(f(c,c),b)$. The automaton recognizing the set of suffixes of the tree $\mathcal{A}_t$ is given in Figure 6.3.*

$Mark(t) = f1(f12(c122, c121), b11)$.

$Q_t = Q_T = \{q_1, q_{12}, q_{11}, q_{122}, q_{121}\}$.

$\Delta_t = \{(c, q_{122}), (c, q_{121}), (b, q_{11}), (f, q_{121}, q_{122}, q_{12}), (f, q_{11}, q_{12}, q_1)\}$.

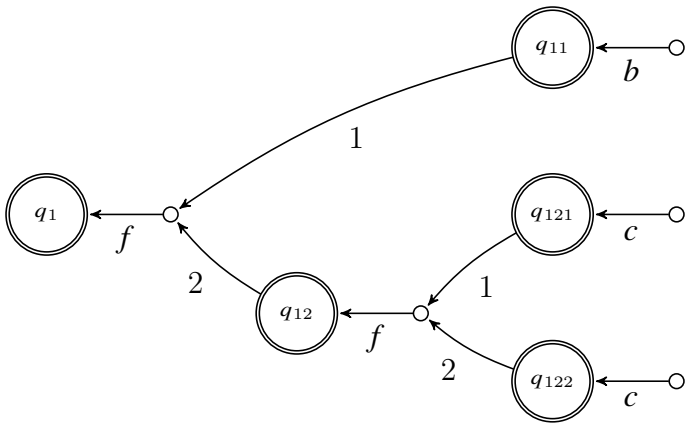


Figure 6.3: Tree suffixes automaton of the pattern $t = f(f(c,c),b)$

The following theorem states that the tree language accepted by a tree suffixes automaton is exactly the set $Suff(t)$.

Theorem 6.4.4 *A tree suffixes automaton $\mathcal{A}_t$ for a tree t recognizes all the suffixes of t.*

In order to prove this theorem, we prove the two next lemmas.

Lemma 6.4.5 *Every suffix of t is recognized by $\mathcal{A}_t$.*

Proof.

Let s be a suffix of the tree t. $s \in Suff(t)$ means that s is:

1. either a leaf, $s \in \Sigma_0$,
2. or $root(s) \in \Sigma_m$ with m the rank of $root(s)$.

The first case is obvious since, by construction, we have transition rules in Δ_t recognizing every leaf symbol. The second case is proved by induction on the structure of the tree t. We consider $s = f(t_1,\ldots,t_m)$ with $f \in \Sigma_m$ and $t_1,\ldots,t_m$ are subtrees of s already recognized by $\mathcal{A}_t$. Then, s is also recognized by $\mathcal{A}_t$ since we have $f(q_{t_1}\ldots,q_{t_m}) \longrightarrow q_f \in \Delta_t$ for each $f \in \Sigma_m$. □

Lemma 6.4.6 *Every tree recognized by $\mathcal{A}_t$ is a suffix of t.*

Proof.

Since all states of the suffixes automaton are final ones we have the two following cases:

1. $\mathcal{A}_t$ recognizes all leaf subtrees as we have transition rules recognizing every leaf symbol $a \in \Sigma_0$.

2. $\mathcal{A}_t$ recognizes subtrees $s = f(t_1,\ldots,t_m)$ with $f \in \Sigma_m$, and $t_1,\ldots,t_m$ are either leaves or have the same structure as s. As we have $f(q_{t_1}\ldots,q_{t_m}) \longrightarrow q_f \in \Delta_t$ for each $f \in \Sigma_m$, then by induction s is a suffix of t.

□

Next, we propose a tree pattern matching algorithm using the tree suffixes automaton to look for fixed tree patterns in a subject tree.

6.4.4 Tree Pattern Matching Algorithm

The principle of the tree pattern matching algorithm using tree suffixes automaton is that if a subtree s of the subject tree t is not a suffix of the pattern to be matched against P, then the whole subtree encapsulating s and having the same height as P, let us call it S, does not match this pattern. In this case, the subtree S is simply pruned or ignored. Otherwise, the subtree S of t is considered as an occurrence of the pattern. In both cases, the tree pattern matching process continues with another subtree, hence the term of sliding or shifting tree. This notion is borrowed from the backward string matching algorithms known as the sliding/shifting window (see [BY89, BM77, CW79, Hor80, Sun90]).

Example 6.4.7 *Let us consider the subject tree t and the tree pattern P of Figure 6.4. Let $S = g(f(b,b))$ be a subtree that has the same height as P. The subtree $s = f(b,b)$ is not a suffix of the pattern P by using automaton of Figure 6.3, thus the whole subtree S can not match the pattern P. Consequently, this subtree is ignored and another subtree is chosen for pattern matching test.*

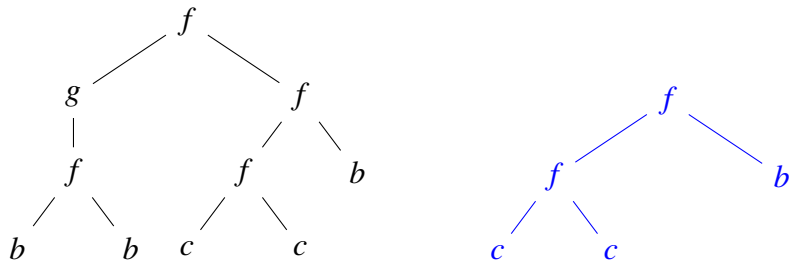


Figure 6.4: Example of subject tree (left) and a fixed tree pattern (right).

We give hereafter the steps of the tree pattern matching using the tree suffixes automaton.

1. Create the bottom-up suffixes tree automaton for the pattern P.

2. Start the pattern matching on the subject tree t using the automaton constructed by selecting a subtree S of t having the same *Height* as P to define the sliding tree.

3. If the matching fails in a node n of S, i.e. the subtree t/n is not a suffix of the tree pattern P. In this case, the subtree in the sliding tree does not match the whole pattern, so it is pruned or ignored.

4. Move the sliding tree to another subtree and restart the tree matching process from step 2.

Selecting a subtree depends on the representation of the subject tree and its implementation.

Algorithm 8: Algorithm *SuffixTPM*

Input: P: Pattern, t: subject tree.

Output: N: Set of nodes.

```
1  Construct the tree suffixes automaton A_P for the pattern P ;
2  N ← ∅;
3  while (t ≠ ∅) do
4      Select a subtree S of t; /* Height(S) = Height(P)        */
5      R ← ∅;
6      foreach (n ∈ S) do
7          if (Δ_P(R,n) ≠ ∅) then
8              R ← R ∪ Δ_P(R,n);
9              if (n = Root(P)) then
10                 N ← N ∪ n;
11         else
12             break;
13     //Prune the subtree S
14     t ← t/S;
15 return N ;
```

6.5 Conclusion

In this chapter we have presented two new algorithms for tree pattern matching problem where we look for one (fixed) or multiple occurrences of trees from some tree language called patterns in a target tree called the subject tree.

In the first approach, we have considered the case of a pattern represented by a regular tree expression. We have presented a tree pattern matching algorithm that runs the extended Thompson automaton on the subject tree.

The tree pattern matching using Thompson's generalization can be done in $O(|t||E|)$ time complexity, where t is the subject tree and E is the regular tree expression representing the pattern language. The novelty of this contribution besides the low time complexity is that the set of patterns can be infinite, since we use regular tree expressions to represent patterns.

In the second approach, we have presented a new solution to the fixed tree pattern matching problem. That is, the looking for a specific single pattern in a subject tree. We have provided a definition for the tree suffixes automaton by generalizing what has been done for strings. The basic idea behind the tree suffix pattern matching is the use of what we have called a sliding tree. This new method of tree pattern matching is much more reliable and less time consuming as we decrease the number of the matching test.

These two pattern matching algorithms have been added to our tree automata toolkit YATAT which will be presented in the next chapter.

Chapter 7

YATAT: Yet Another Tree Automata Toolkit

Tree automata are a powerful theoretical tool used in many fields. Many algorithms dedicated to solve problems related to tree automata have been proposed in the literature. However, few toolkits have been implemented in order to manipulate, evaluate or compare tree automata algorithms. In this chapter, we describe an extension of the YATAT (Yet Another Tree Automata Toolkit) Toolkit that implements algorithms for construction and manipulation of tree automata. This toolkit has been initially developed by Guellouma as a part of his PhD thesis [Gue17] and a South Africa-Algeria Cooperation Project[1].

Our objective through the development of this toolkit has been to study the existing tree automata algorithms and then, to implement these algorithms in a toolkit in order to facilitate the manipulation of trees, automata, regular expressions with the underlying algorithms. In this version of our toolkit, we have focused on two categories of algorithms: minimization and conversion from rational tree expressions to finite tree automata and back. As for applications, we have implemented some minimization algorithms and our two tree pattern matching algorithms proposed in this thesis.

The novelty of this toolkit resides in its modularity and data-structure-free-dependence, which makes it easily expandable to new algorithms for a large number of applications. We have used the D programming language [Ale10] to implement basic modules and the standard XML (TIMBUK) files as input/output data structures.

7.1 Tree Automata Toolkits: Related work

Most of the tree automata techniques and algorithms are a generalization of what has been previously done on strings. However, compared to the later, there is a lack of toolkits and software platforms for tree automata.

The first well-known platform using tree automata is MONA for Monadic Second-Order Logic in Practice [EKM98]. It is dedicated to logic formulas checking. The second most famous toolkit dealing with tree automata is TIMBUK [GT01]. It is a collection of tools and data structures dedicated to proofs of reachability over Term Rewriting Systems (based on another tree automata toolkit called ELAN [BKK+96]) for manipulating Tree Automata (bottom-up nondeterministic finite tree automata). It has been implemented in Ocaml. Taml is another important toolkit

[1]This work was supported by a South Africa-Algeria Cooperation Project funded by the South African National Research Foundation and the Algerian MESRS-DGRSDT under project A/AS-2013-002.

that implements the basic operations of tree automata manipulations equipped with Tabi, its graphical interface. Another toolkit based on TIMBUK is TIBURON [MK06], it has been developed especially for weighted tree automata. LIBVATA [LSV12] is a recent TA toolkit oriented to verification techniques using tree automata. Finally, ForestFire [CH09] is a taxonomy based toolkit for tree algorithms related to acceptance and tree pattern matching problems. The following table summarizes the most important toolkits along with their implemented functionalities and availability.

Toolkit	Automata	Algorithms	Availability
MONA [EKM98] (1998)	Bottom-up tree automata	Translates formulas to finite-state automata.	On `http://www.brics.dk/mona`
TIMBUK [GT01] (2001)	bottom-up non-deterministic tree automata	Collection of tools for achieving proofs of reachability over Term Rewriting Systems and for manipulating Tree Automata	On `http://www.irisa.fr/lande/genet/timbuk`
AutoWrite [Dur05] (2004)	bottom-up tree automata	Checking properties of term rewrite systems and handling tree automata	On `http://dept-info.labri.fr/~idurand/autowrite`
TIBURON [MK06] (2006)	Weighted tree automata	Weighted regular tree grammars, context-free grammars, string transducers, determinization, composition, intersection, ...	On `http://www.isi.edu/licensed-sw/tiburon`
ForestFire [CH09] (2009)	Top-down, Bottom-up, NFTA	Acceptance, pattern matching, and parsing	Not available on line
LIBVATA [LSV12] (2012)	bottom-up non-deterministic tree automata	Formal verification with tree automata	On `http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/`
TALib [HMQ13] (2013)	Weighted tree automata	Construction and manipulation of tree automata essentially minimization and equivalence	Not available on line

Table 7.1: Details on existing TA toolkits

7.2 Description of the Toolkit

YATAT is a tree automata toolkit that has been designed for an easy implementation of regular tree languages' algorithms. YATAT core has been written essentially in *D* programming language [Ale10], which allows to exploit the efficiency and the originality of this language. However, some code fragments have been written in C++ and Java. This ensures the toolkit to run on multiple platforms, namely Windows and Linux [Gue17]. Hereafter, we recall the basic architecture of YATAT and the main data structures used.

7.2.1 YATAT Core

For the internal architecture, the toolkit is build around a set of independent *Modules* inheriting from a single meta module. Each one of them implements a particular set of functions or an algorithm. This gives a better fluidity to the toolkit and an easier way to incorporate new add-ons. Some extra modules implementing some basic operations on tree automata like reduction and determinization are also used.

Figure 7.1 shows YATAT principal modules. As underlined earlier, YATAT is extensible and allows the addition of new modules. The main concept behind this idea is the use of inter-operable format for trees, tree automata, ... representation. In fact, we have opted for the standard XML format (see Figure 7.2 for example). Thus, instead of proposing a unified data structure of tree automata, a specific constructor from the XML file is used for each module.

For modules that modify tree automata, changes are made only through XML format file and consequently, the other algorithms can be safely used. Figure 7.3 presents the UML class diagram of YATAT showing its main classes and the different relations between them.

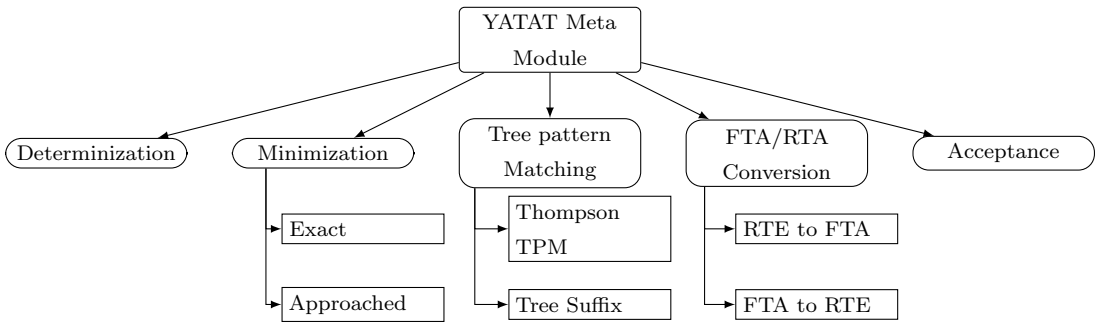


Figure 7.1: YATAT main modules

```
<?xml version="1.0"?>
<automaton>
<nature>Ranked,Unweighted</nature>
<auto>
<alphabet>a,0;b,0;g,1;f,2;</alphabet>
<states>q1,q2,q3,q4,</states>
<Fstates>q3,q4,</Fstates>
<delta>
        <rule>a,q1,</rule>
        <rule>b,q2,</rule>
        <rule>g,q1,q3,</rule>
        <rule>g,q2,q4,</rule>
        <rule>f,q2,q4,q4,</rule>
        <rule>f,q1,q3,q4,</rule>
        <rule>f,q2,q3,q4,</rule>
        <rule>f,q1,q4,q4,</rule>
</delta>
</auto>
</automaton>
```

Figure 7.2: Example of a tree automaton in XML.

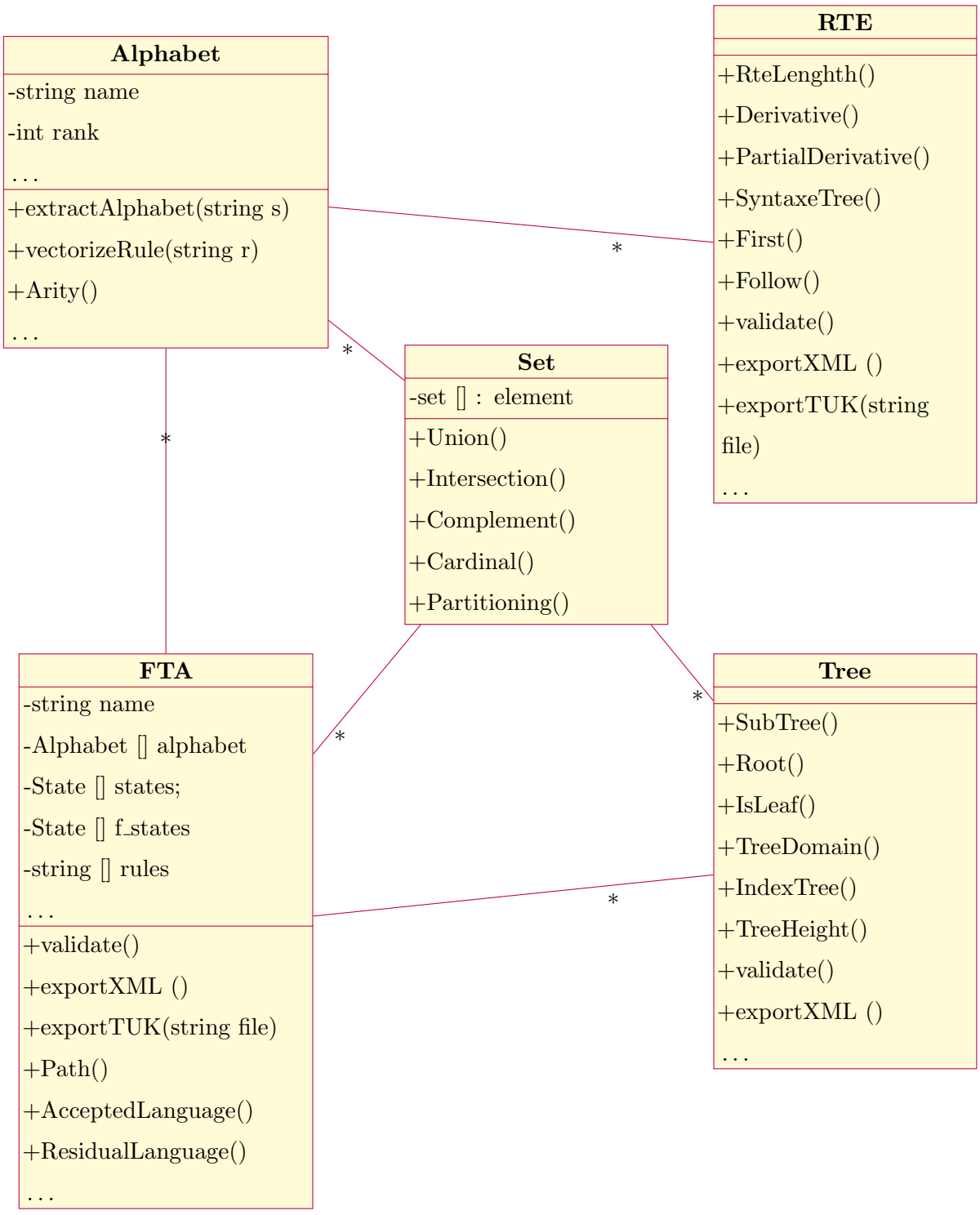


Figure 7.3: YATAT main classes.

7.2.2 Using YATAT Toolkit

YATAT might be used with both command line and graphical versions. To use the command line version of YATAT, one would have to run the toolkit's script with XML/TIMBUUK files as input parameters. The result might be printed on the screen or written in an output XML file. For example, for the tree pattern matching

algorithm using Thompson's generalization test, we have run the YATAT script with two XML files as input: one for the regular tree expression of the pattern, and another for the subject tree. Figures 7.4,7.5, 7.6, 7.7 show some results after running YATAT.

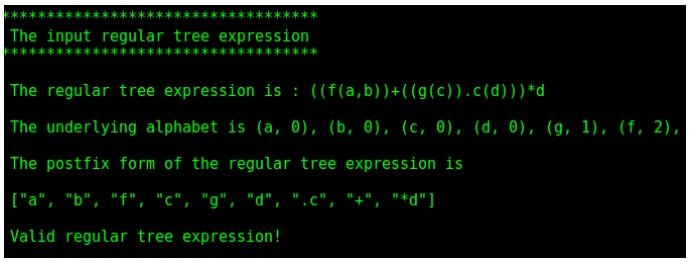


Figure 7.4: Example of validating a regular tree expression with YATAT

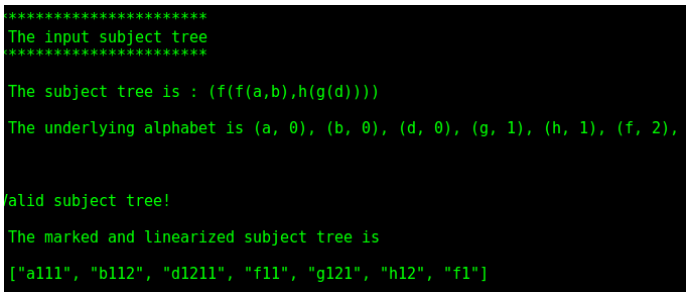


Figure 7.5: Example of validating a subject tree with YATAT

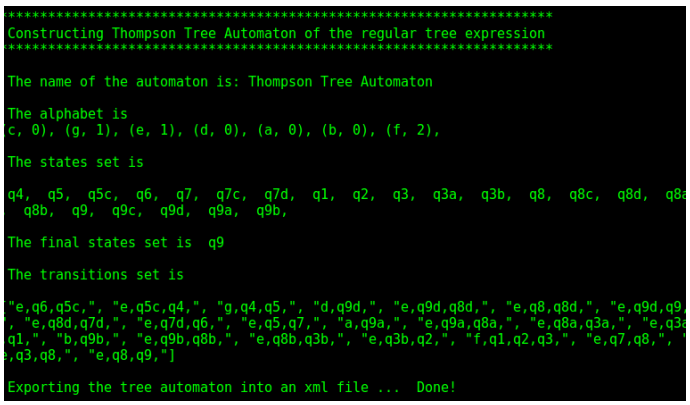


Figure 7.6: Example of a Thomspson tree automaton construction with YATAT

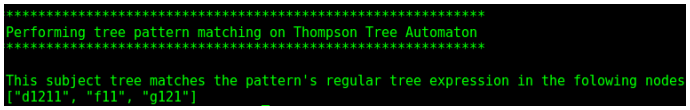


Figure 7.7: Example of tree pattern matching with YATAT

YATAT is publicly available on Bitbuckt which is a a web-based version control repository hosting service owned by Atlassian, for source code and development projects that use either Mercurial or Git version control systems[2].

7.3 Tree Automata Algorithms Implemented in YATAT

Before implementing YATAT, we have started by making an inventory of the different tree automata and regular tree expressions existing algorithms in basically two categories: minimization and conversion between tree automata and regular tree expressions. This has been done by taking into account the following criteria :

- Type of the tree automata: top-down vs bottom-up, acyclic vs cyclic, weighted vs unweighted, ...
- Type of the tree: Ranked vs unranked.
- Nature of the algorithm: sequential vs parallel.
- Function involving data structures: manipulation or application.
- The main idea of the problem resolution.

Hereafter, we present the two categories of tree automata algorithms implemented by now in our toolkit YATAT.

7.3.1 Minimization Algorithms

According to the criteria above, we first classify minimization techniques with respect to their ability to preserve the language and the structure of the input automaton. A technique is said to be *exact* if the language of the automaton after minimization is the same and if the resulting automaton is the minimal one we can obtain. However, an *approached* technique is a minimization technique that results an automaton which does not preserve the language (such as in the Hyper-minimization) or if it is not minimal. This criterion is considered after a distinction between techniques, according to the automaton nature. This part of the toolkit has been implemented by Guellouma [Gue17].

[2]Available on `https://bitbucket.org/AhlemBelabbaci/yatat/src/master/`

7.3.2 Conversion of TA to RTE and back algorithms

This part of the toolkit deals with different algorithms for the proof of Kleene theorem for trees. Figure 7.8 shows the algorithms implemented in YATAT for both conversions: from tree automata to regular expressions and back.

7.4 Conclusion

In this chapter we have described the general framework of our tree automata toolkit YATAT. We have started by giving an overview of the existing toolkits with outlines of their main features. Then, we have presented our extension to this toolkit. The ongoing version of YATAT implements two categories of algorithms: one for tree automata minimization and the other for the algorithms of the conversion of tree automata to regular expressions and back. As for applications, two tree pattern matching algorithms have been implemented with a customized random tree generator for experimentations. As we aimed to provide a simple, intuitive, and an expendable toolkit, YATAT is based on many inter-operable modules using XML/TIMBUK standard files for input/output data structures. For a medium-term perspective, we intend to implement more tree automata algorithms and tree automata related applications in YATAT.

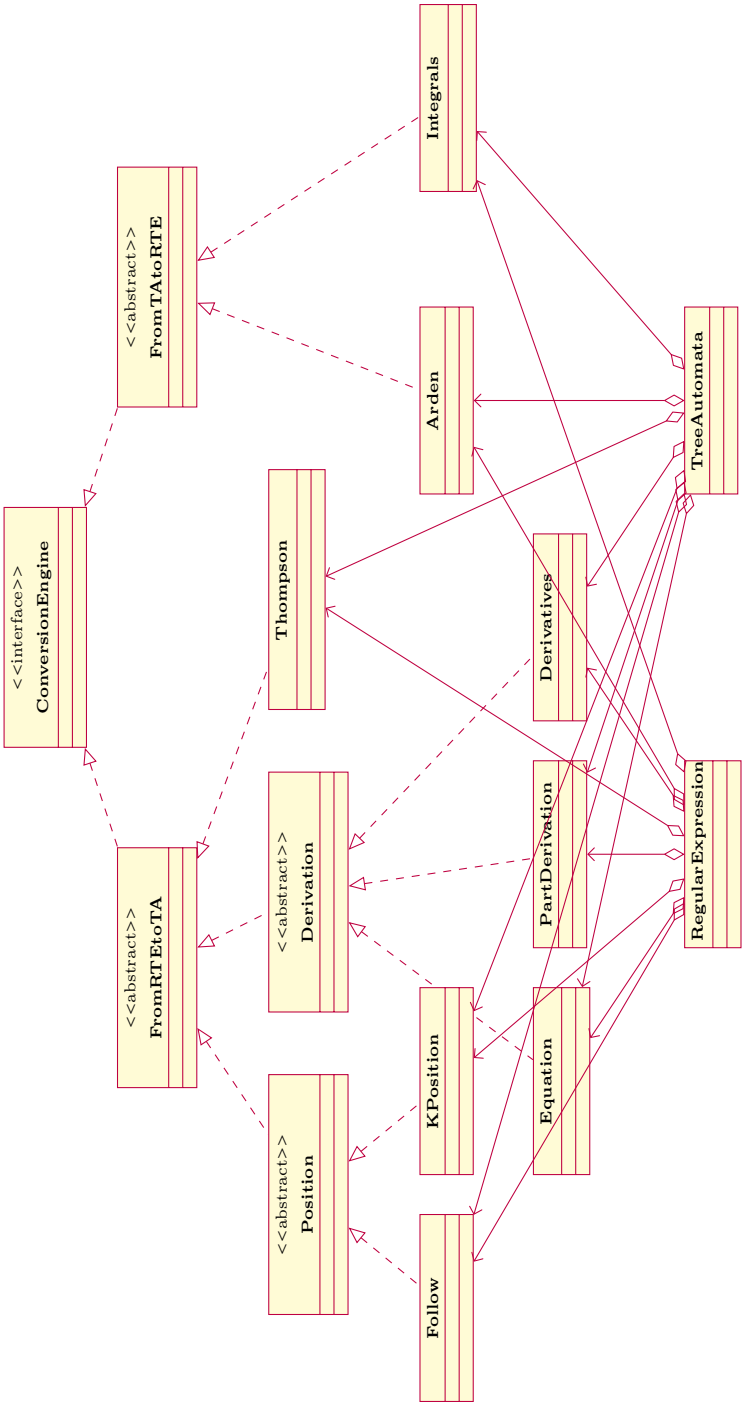


Figure 7.8: Hierarchy of conversion algorithms implemented in YATAT

Conclusion and Perspectives

In this thesis, we have studied two important aspects of regular tree languages: proof of Kleene theorem for trees and tree pattern matching problem.

In the part dealing with Kleene theorem for trees, that is the equivalence between tree automata and regular tree expressions, we have investigated the conversions between these two formalisms. We have studied the different approaches in this context and then proposed two new algorithms:

i. In the conversion from regular tree expressions to tree automata, a generalization of the well-known Thompson construction from strings to trees [BCZC15, BCCZ18]. This generalization allows an inductive construction of a tree automaton from a regular tree expression. We have proposed a general form of Thompson's tree automaton that keeps the same characteristics of Thompson's automaton for strings.

ii. The conversion back is also another generalization from strings to trees of the integrals [GBC15]. It allows the synthesis of a regular tree expression from a tree automaton. We have defined integrals for trees, which are considered as the inverse operation of derivatives. This generalization is used to denote by a regular tree expression, the accepted language of a tree automaton.

The other part of the thesis has been devoted to the tree pattern matching problem. We have given a brief literature overview of the existing algorithms and then, we have proposed two new algorithms:

i. In the first one, the pattern to be matched against is represented by a regular tree expression. The pattern matching is achieved using the Thompson's generalization for trees [BCCZ18]. This algorithm requires $O(|t||E|)$ time complexity, where $|t|$ is the size of the subject tree and $|E|$ is the size of the regular tree expression of the tree pattern. The novelty of this contribution besides the low time complexity is that the set of patterns can be infinite, since we use regular tree expressions to represent patterns.

ii. The other algorithm deals rather with the fixed tree pattern matching problem. Here, we have generalized the well-known structure of factor oracle from strings to trees. We have proposed an algorithm looking for fixed tree patterns in a subject tree. This algorithm uses what we have called sliding tree by analogy to sliding windows in the string case.

The last part of the thesis has been dedicated to the presentation and extension of the tree automata toolkit YATAT. All the algorithms of conversion between regular tree expressions and tree automata, and tree pattern matching, studied in the present thesis, have been added to YATAT.

Despite the low theoretical complexity of our tree pattern matching algorithms, it is necessary to get an idea about its practical one, which we estimate to be much lower since we have bounded all subsets of states by the set of all states Q. In order to do that, we have implemented besides the tree pattern matching algorithm and the tree automaton acceptor, a random generator of regular tree expressions and subject trees.

The random generator in question must be a parametrized one, that is, it generates random regular tree expressions as patterns and random subject trees such that the latter should contain at least one occurrence of the underlying pattern so we can test our tree pattern matching algorithm.

We have proposed an implementation of a parametrized generator in order to test our first tree pattern matching algorithm. However, we have been unable to compare our results with other tree pattern matching algorithms because of the difference of the input parameters. In other words, our approach is the only one that represents the tree pattern by a regular tree expression.

Perspectives

This work might be further developed in terms of:

- Deepen our study on the proposed Thompson tree automata, more precisely on the structure of the automaton and its characterization as it has been done for word [GPWZ04].

- Generalizing the algorithms proposed to other hierarchy in tree languages for example the context free ones, and other tree automata like tree transducers, weighted tree automata and hedge automata.

- For tree pattern matching problem, generalizing more algorithms of string matching to trees, especially the ones that have been proved with the lowest time complexity.
- Enhance the algorithm of the fixed tree pattern matching in order to look for patterns with variable leaves or multiple patterns in the subject tree instead of just a specified pattern.
- For the random generation of customized subject trees and tree patterns, we have to propose and prove a probability distribution that assigns a probability to each subset of possible patterns that might appear in a such a random generation.
- Extending YATAT toolkit by implementing more algorithms and applications related to regular tree languages.

Bibliography

[AA04] Tony Abou-Assaleh and Wei Ai. Survey of global regular expression print (grep) tools. Technical report, 2004.

[ACR99] Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Oracle des facteurs, oracle des suffixes. Technical report, Institut Gaspard-Monge, Université de Marne-la-Valléee, 1999.

[ACT10] Andrea Asperti, Claudio Sacerdoti Coen, and Enrico Tassi. Regular expressions, au point. *CoRR*, abs/1010.2604, 2010.

[AD04] Gérard Assayag and Shlomo Dubnov. Using factor oracles for machine improvisation. *Soft Computing*, 8(9):604–610, 2004.

[AG85] Alfred V. Aho and Mahadevan 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*, pages 334–340. ACM, 1985.

[AH74] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[Ale10] Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley Professional, 1st edition, 2010.

[Ant96] Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[Ard61] Dean N Arden. Delayed-logic and finite-state machines. In *Switching Circuit Theory and Logical Design, 1961. SWCT 1961. Proceedings of the Second Annual Symposium on*, pages 133–151. IEEE, 1961.

[Atk89] Kendall E Atkinson. *An introduction to numerical analysis. 1989*. John Wiley & Sons, 1989.

[BCCZ18] Ahlem Belabbaci, Hadda Cherroun, Loek Cleophas, and Djelloul Ziadi. Tree pattern matching from regular tree expressions. *Kybernetika*, 54(2):221–242, 2018.

[BCZC15] Ahlem Belabbaci, Hadda Cherroun, Djelloul Ziadi, and Loek Cleophas. Construction of Thompson's automaton from a regular tree expression. In *3rd International Workshop on Trends in Tree Automata and Tree Transducers TTATT, at London UK*, 2015.

[Bel14] Ahlem Belabbaci. Recherche de motifs d'arbre. Master's thesis, Université Amar Telidji, Laghouat, 2014.

[Ber05] Jean Berstel. Support de cours, automates et grammaires. Université de Marne-la-Vallée, France, 2004-2005.

[BK93] Anne Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[BKK+96] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. *Electr. Notes Theor. Comput. Sci.*, 4:35–50, 1996.

[BKMW01] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular tree and regular hedge languages over unranked alphabets. Technical report, 2001.

[BM63] Janusz A Brzozowski and Edward J McCluskey. Signal flow graph techniques for sequential circuit state diagrams. *IEEE Transactions on Electronic Computers*, (2):67–76, 1963.

[BM77] Robert S Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[Bra69] Walter S. Brainerd. Tree generating regular systems. *Information and Control*, 14(2):217 – 231, 1969.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[BS86] Gerard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical computer science*, 48:117–126, 1986.

[BY89] Ricardo A Baeza-Yates. Improved string searching. *Software: Practice and Experience*, 19(3):257–271, 1989.

[CCG+94] Maxime Crochemore, Artur Czumaj, Leszek Gasieniec, Stefan Jarominek, Thierry Lecroq, Wojciech Plandowski, and Wojciech Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12(4-5):247–267, 1994.

[CDG+08] H. Comon, M. Dauchet, R. Gilleron, C. Loding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2008.

[CH09] Loek G. Cleophas and Kees Hemerik. Forest FIRE: A taxonomy-based toolkit of tree automata and regular tree algorithms. In *Implementation and Application of Automata, 14th International Conference, CIAA 2009, Sydney, Australia, July 14-17, 2009. Proceedings*, pages 245–248, 2009.

[Cle08] Loek Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit.* PhD thesis, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, 2008.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[CMM99] Allauzen Cyril, Crochemore Maxime, and Raffinot Mathieu. Factor oracle: A new structure for pattern matching. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 295–310. Springer, 1999.

[CMM01] Allauzen Cyril, Crochemore Maxime, and Raffinot Mathieu. Efficient experimental string matching by weak factor recognition. In *Annual Symposium on Combinatorial Pattern Matching*, pages 51–72. Springer, 2001.

[Con12] John Horton Conway. *Regular algebra and finite machines.* Courier Corporation, 2012.

[Cou78] Bruno Courcelle. A representation of trees by languages. *Theoretical Computer Science*, 6(3):255 – 279, 1978.

[Cox11] Russ Cox. Implementing regular expressions. Technical report, 2011.

[CP92] Chia-Hsiang Chang and Robert Paige. From regular expressions to DFA's using compressed NFA's. In *Annual Symposium on Combinatorial Pattern Matching*, pages 90–110. Springer, 1992.

[CW79] Beate Commentz-Walter. A string matching algorithm fast on the average. In *International Colloquium on Automata, Languages, and Programming*, pages 118–132. Springer, 1979.

[CZ01] Jean-Marc Champarnaud and Djelloul Ziadi. From C-Continuations to new quadratic algorithms for automaton synthesis. *IJAC*, 11(6):707–736, 2001.

[CZ02] Jean-Marc Champarnaud and Djelloul Ziadi. Canonical derivatives, partial derivatives and finite automaton constructions. *Theor. Comput. Sci.*, 289(1):137–163, 2002.

[CZW03] Loek G. Cleophas, Gerard Zwaan, and Bruce W. Watson. Constructing factor oracles. In *Stringology*, 2003.

[DDP+05] Jean-François Dufayard, Laurent Duret, Simon Penel, Manolo Gouy, François Rechenmann, and Guy Perrière. Tree pattern matching in phylogenetic trees: automatic search for orthologs or paralogs in homologous gene sequence databases. *Bioinformatics*, 21(11):2596–2603, 2005.

[DH03] Frank Drewes and Johanna Högberg. Learning a regular tree language from a teacher. In *Developments in Language Theory, 7th International Conference*, pages 279–291, 2003.

[DKV09] Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of weighted automata*. Springer Science & Business Media, 2009.

[Don71] John Doner. Tree acceptors and some of their applications. *J. Comput. Syst. Sci.*, 5(4), 1971.

[Dre09] Frank Drewes. Lecture notes on Tree Automata, Umea University, Department of Computing Science, Sweden, 2009.

[Dur05] Irène Durand. Autowrite: A tool for term rewrite systems and tree automata. *Electronic Notes in Theoretical Computer Science*, 124(2):29–49, 2005.

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: new techniques for WS1S and WS2S. In *Proc. 10th International Conference on Computer-Aided Verification (CAV)*, volume 1427 of *LNCS*, pages 516–520. Springer-Verlag, 1998.

[Eng15] Joost Engelfriet. Tree automata and tree grammars. *arXiv preprint arXiv:1510.02036*, 2015.

[FHP92] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. Burg: Fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, 1992.

[FIJ+12] Tomáš Flouri, Costas S. Iliopoulos, Jan Janousek, Borivoj Melichar, and Solon P. Pissis. Tree template matching in ranked ordered trees by pushdown automata. *J. Discrete Algorithms*, 17:15–23, 2012.

[Flo12] Tomáš Flouri. *Pattern Matching in Tree Structures*. PhD thesis, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University, 2012.

[FV09] Zoltán Fülöp and Heiko Vogler. Weighted tree automata and tree transducers. In *Handbook of Weighted Automata*, pages 313–403. Springer, 2009.

[GBC15] Younes Guellouma, Ahlem Belabbaci, and Hadda Cherroun. Towards integrals of rational tree expressions. In *Conference on Discrete Mathematics and Computer Science, Sidi Bel Abbess, Algeria*, pages 151–154, 2015.

[GH15] Hermann Gruber and Markus Holzer. From finite automata to regular expressions and back: A summary on descriptional complexity. *International Journal of Foundations of Computer Science*, 26(08):1009–1040, 2015.

[Gie98] Robert Giegerich. A declarative approach to the development of dynamic programming algorithms, applied to RNA folding. *Report 98*, 2, 1998.

[GK00] Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In *Proceedings of the 17th International Conference on Automated Deduction*, CADE-17, pages 271–290, London, UK, 2000. Springer-Verlag.

[GL00] Jean Goubault-Larrecq. A method for automatic cryptographic protocol verification. *Parallel and Distributed Processing*, pages 977–984, 2000.

[GLRÁ11] Pedro García, Damián López, José Ruiz, and Gloria I Álvarez. From regular expressions to smaller NFAs. *Theoretical Computer Science*, 412(41):5802–5807, 2011.

[Glu61] Victor M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961.

[GMCZ15] Younes Guellouma, Ludovic Mignot, Hadda Cherroun, and Djelloul Ziadi. Construction of rational expression from tree automata using a generalization of Arden's lemma. *CoRR*, abs/1501.07686, 2015.

[Goe95] Eric Goebelbecker. Using grep: Moving from DOS? Discover the power of this Linux utility. *Linux Journal*, 18, 1995.

[GPWZ04] Dora Giammarresi, Jean-Luc Ponty, Derick Wood, and Djelloul Ziadi. A characterization of Thompson digraphs. *Discrete Applied Mathematics*, 134(1-3):317–337, 2004.

[Grä91] Albert Gräf. Left-to-right tree pattern matching. In *Rewriting Techniques and Applications*, pages 323–334. Springer, 1991.

[GS97] Ferenc Gécseg and Magnus Steinby. Handbook of formal languages, vol. 3. chapter Tree Languages, pages 1–68. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[GS15] Ferenc Gécseg and Magnus Steinby. Tree automata. *arXiv preprint arXiv:1509.06233*, 2015.

[GT01] Thomas Genet and Valérie Viet Triem Tong. Reachability analysis of term rewriting systems with Timbuk. In *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*, pages 695–706, 2001.

[Gue83] Irène Guessarian. Pushdown tree automata. *Mathematical Systems Theory*, 16(4):237–263, 1983.

[Gue17] Younes Guellouma. *Algorithmique des automates d'arbres en vue d'un traitement efficace des grandes masses de données*. PhD thesis, Université Amar telidji, Laghouat, 2017.

[Har78] M. A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1978.

[HMQ13] Thomas Hanneforth, Andreas Maletti, and Daniel Quernheim. Random generation of nondeterministic finite-state tree automata. *arXiv preprint arXiv:1311.5568*, 2013.

[HMV09] Johanna Högberg, Andreas Maletti, and Heiko Vogler. Bisimulation minimisation of weighted automata on unranked trees. *Fundamenta Informaticae*, 92(1-2):103–130, 2009.

[HO82] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, 1982.

[Hög07] Johanna Högberg. *Contributions to the Theory and Applications of Tree Languages*. PhD thesis, Department of Computing Science, Umeå University, 2007.

[Hop08] John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 2008.

[Hor80] R Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

[HSW01] Juraj Hromkovič, Sebastian Seibert, and Thomas Wilke. Translating regular expressions into small ε-free nondeterministic finite automata. *Journal of Computer and System Sciences*, 62(4):565–588, 2001.

[IWIU12] Yuko Itokawa, Masanobu Wada, Toshimitsu Ishii, and Tomoyuki Uchida. Pattern matching algorithm using a succinct data structure for tree-structured patterns. In *Intelligent Control and Innovative Computing*, volume 110 of *Lecture Notes in Electrical Engineering*, pages 349–361. Springer US, 2012.

[IY03] Lucian Ilie and Sheng Yu. Follow automata. *Information and computation*, 186(1):140–162, 2003.

[Kle56] Stephen C Kleene. *Automata Studies*. Princeton University Press, 1956.

[KM11] Dietrich Kuske and Ingmar Meinecke. Construction of tree automata from regular expressions. *RAIRO - Theor. Inf. and Applic.*, 45(3):347–370, 2011.

[KMP77] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[KOZ08] Ahmed Khorsi, Faissal Ouardi, and Djelloul Ziadi. Fast equation automaton computation. *Journal of Discrete Algorithms*, 6(3):433–448, 2008.

[Kro75] Hans Hermann Kron. *Tree Templates and Subtree Transformational Grammars*. PhD thesis, University of California, Santa Cruz, 1975.

[Kui01] Werner Kuich. Pushdown tree automata, algebraic tree systems, and algebraic tree series. *Information and Computation*, 165(1):69–99, 2001.

[KW05] Ryoichi Kato and Osamu Watanabe. Substring search and repeat search using factor oracles. *Information Processing Letters*, 93(6):269–274, 2005.

[Lev79] Barry Levine. *The automated inference of tree system*. PhD thesis, Oregon State University, 1979.

[Lev81] Barry Levine. Derivatives of tree sets with applications to grammatical inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3(3):285–293, 1981.

[Lev82] Barry Levine. The use of tree derivatives and a sample support parameter for inferring tree systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(1):25–34, 1982.

[Lib06] Leonid Libkin. Logics for unranked trees: An overview. *arXiv preprint cs/0606062*, 2006.

[LL02a] Arnaud Lefebvre and Thierry Lecroq. Compror: on-line lossless data compression with a factor oracle. *Information Processing Letters*, 83(1):1–6, 2002.

[LL02b] Arnaud Lefebvre and Thierry Lecroq. A heuristic for computing repeats with a factor oracle: application to biological sequences. *International Journal of Computer Mathematics*, 79(12):1303–1315, 2002.

[LLA03] Arnaud Lefebvre, Thierry Lecroq, and Joël Alexandre. An improved algorithm for finding longest repeats with a modified factor oracle. *Journal of Automata, Languages and Combinatorics*, 8(4):647–657, 2003.

[Löd12] Christof Löding. Basics on tree automata. In *Modern Applications of Automata Theory*, pages 79–110. 2012.

[LSV12] Ondrej Lengál, Jirí Simácek, and Tomás Vojnar. VATA: A library for efficient manipulation of nondeterministic tree automata. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 79–94, 2012.

[LSZ13] Éric Laugerotte, Nadia Ouali Sebti, and Djelloul Ziadi. From regular tree expression to position tree automaton. In *Language and Automata Theory and Applications - 7th International Conference, LATA 2013, Bilbao, Spain, April 2-5, 2013. Proceedings*, pages 395–406, 2013.

[LT14] Laks VS Lakshmanan and Alex Thomo. View-based tree language rewritings for XML. In *International Symposium on Foundations of Information and Knowledge Systems*, pages 270–289. Springer, 2014.

[LW00] Hsiao-Tzu Lu and Yang Wuu. A simple tree pattern-matching algorithm. In *Proceedings of the Workshop on Algorithms and Theory of Computation*. Citeseer, 2000.

[Mal05] Andreas Maletti. The power of tree series transducers of type i and ii. In *International Conference on Developments in Language Theory*, pages 338–349. Springer, 2005.

[MAMO69] M M Agidor and G M Oran. Finite automata over finite trees. Technical report, Hebrew University, Jerusalem, 1969.

[MK06] Jonathan May and Kevin Knight. Tiburon: A weighted tree automata toolkit. In *Implementation and Application of Automata, 11th International Conference, CIAA 2006, Taipei, Taiwan, August 21-23, 2006, Proceedings*, pages 102–113, 2006.

[MM01] Panagiotis D Michailidis and Konstantinos G Margaritis. On-line string matching algorithms: Survey and experimental results. *International journal of computer mathematics*, 76(4):411–434, 2001.

[MM05] Alban Mancheron and Christophe Moan. Combinatorial characterization of the language recognized by factor and suffix oracles. *International Journal of Foundations of Computer Science*, 16(06):1179–1191, 2005.

[MNS08] Wim Martens, Frank Neven, and Thomas Schwentick. Deterministic top-down tree automata: past, present, and future. In *Logic and Automata: History and Perspectives.*, pages 505–530, 2008.

[MS98] Maya Madhavan and Priti Shankar. Optimal regular tree pattern matching using pushdown automata. In *Foundations of Software Technology and Theoretical Computer Science, 18th Conference, Chennai, India, December 17-19, 1998, Proceedings*, pages 122–133, 1998.

[MSZ14a] Ludovic Mignot, Nadia Ouali Sebti, and Djelloul Ziadi. An efficient algorithm for the equation tree automaton via the k-c-continuations. In *Language, Life, Limits - 10th Conference on Computability in Europe, CiE 2014, Budapest, Hungary, June 23-27, 2014. Proceedings*, pages 303–313, 2014.

[MSZ14b] Ludovic Mignot, Nadia Ouali Sebti, and Djelloul Ziadi. k-position, follow, equation and k-C-Continuation tree automata constructions. In *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014.*, pages 327–341, 2014.

[MSZ17] Ludovic Mignot, Nadia Ouali Sebti, and Djelloul Ziadi. Tree automata constructions from regular expressions: a comparative study. *Fundam. Inform.*, 156(1):69–94, 2017.

[Mur99] Makoto Murata. Hedge automata: a formal model for XML schema, 1999.

[MY60] R. McNaughton and H. Yamada. Regular expressions and finite state graphs for automata. *Electronic Computers, IRE Transactions on*, EC-9(1):39–47, 1960.

[NR98] Gonzalo Navarro and Mathieu Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Annual Symposium on Combinatorial Pattern Matching*, pages 14–33. Springer, 1998.

[OF61] Gene Ott and Neil H. Feinstein. Design of sequential machines from their regular expressions. *J. ACM*, 8(4):585–600, 1961.

[OW07] Andy Oram and Greg Wilson. *Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice (O'Reilly))*. O'Reilly Media, Inc., 2007.

[PJM11] Radomír Polách, Jan Janoušek, and Borivoj Melichar. Regular tree expressions and deterministic pushdown automata. In *Mathematical and Engineering Methods in Computer Science - 7th International Doctoral Workshop, MEMICS 2011, Lednice, Czech Republic*, pages 70–77, 2011.

[Rab68] Michael O Rabin. Decidability of second-order theories and automata on infinite trees. *Bulletin of the American Mathematical Society*, 74(5):1025–1029, 1968.

[Rou70] William C. Rounds. Mappings and grammars on trees. *Mathematical systems theory*, 4:257–287, 1970.

[RS59] Michael O Rabin and Dana Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

[RS97] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages, Vol. 1: Word, Language, Grammar*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[RUG97] Edward M. Reingold, Kenneth J. Urban, and David Gries. K-M-P string matching revisited. *Inf. Process. Lett.*, 64(5):217–223, 1997.

[Sal88] Kai Salomaa. Deterministic tree pushdown automata and monadic tree rewriting systems. *Journal of Computer and System Sciences*, 37(3):367–394, 1988.

[SG85] Karl M Schimpf and Jean H Gallier. Tree pushdown automata. *Journal of Computer and System Sciences*, 30(1):25–40, 1985.

[ST09] Jacques Sakarovitch and Reuben Thomas. *Elements of automata theory*. Cambridge University Press, 2nd edition, 2009.

[Sun90] Daniel M Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990.

[SY72] L. W. Smith and Stephen S. Yau. Generation of regular expressions for automata by the integral of regular expressions. *Comput. J.*, 15(3):222–228, 1972.

[Tha70] James W. Thatcher. Generalized sequential machine maps. *J. Comput. Syst. Sci.*, 4(4):339–367, 1970.

[Tha73] James. W. Thatcher. Tree automata: An informal survey. In *A. V. Aho, ed., Currents in the Theory of Computing*, page 143172. Prentice Hall, New York, 1973.

[Tho68] Ken Thompson. Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[TJMC15] Jan Trávnícek, Jan Janousek, Borivoj Melichar, and Loek G. Cleophas. Backward linearised tree pattern matching. In *Language and Automata Theory and Applications - 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*, pages 599–610, 2015.

[TW68] James W. Thatcher and Jesse B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical systems theory*, 2(1):57–81, 1968.

[Wat95] Bruce W Watson. Taxonomies and toolkits of regular language algorithms. *PhD. Eindhoven University of Technology, Netherlands*, 1995.

[Woo87] Derick Wood. *Theory of computation*. Wiley, 1987.