FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER’S THESIS

Title:	Finite tree automata to regular tree expressions conversion by removal of states
Student:	Bc. Tomáš Dejmek
Supervisor:	Ing. Jan Trávníček
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2020/21

Instructions

Study the definition of finite automata and regular expressions.
Study the finite automaton to regular expression conversion by removal of states.
Study the definition of finite tree automata and tree regular expressions.
Design a new algorithm for the finite tree automaton to tree regular expression conversion by removal of states.
Implement the finite tree automaton to tree regular expression conversion by removal of states.
Propose suitable tests and test your implementation.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 24, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

Finite tree automata to regular tree expressions conversion by removal of states

Bc. Tomáš Dejmek

Department of Computer Science
Supervisor: Ing. Jan Trávníček, Ph.D.

May 9, 2019

Acknowledgements

I would like to thank my supervisor Ing. Jan Trávníček, Ph.D. for this assignment and valuable advising. I would also like to thank the Faculty of Information Technology for the acquired knowledge, especially Ing. Ladislav Vagner, Ph.D., Ing. Tomáš Valla, Ph.D. and Ing. Martin Kačer, Ph.D. that they helped me to develop problem-solving skills, which I would never reach alone. Last but not least I would like to thank my family and colleagues for their support.

Declaration

In Prague on May 9, 2019 . . . .

Citation of this thesis

Dejmek, Tomáš. *Finite tree automata to regular tree expressions conversion by removal of states.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Tato diplomová práce se zabývá stromovými jazyky. Stromový jazyk se skládá z množiny stromů, podobně jako textový jazyk se skládá z množiny textových řetězců. Jelikož byla dokázána rovnost mezi třídou jazyků přijímaných konečnými stromovými automaty a třídou jazyků popisovanou regulárními stromovými výrazy, tak je možné provádět převody. Já se zaměřují na převod konečných stromových automatů na regulární stromové výrazy. Během práce je představeno jedno současné řešení a další dva nové přístupy jsou navrženy a implementovany. Práce také zahrnuje porovnání nových algoritmů včetně měření zavedené kvality výsledku.

Klíčová slova konečný stromový automat, regularní stromový výraz, konverze

Abstract

This thesis studies tree languages. Tree language is composed from set of trees similarly as textual language is composed from set of textual strings. Since class of languages which are accepted by finite tree automata and class of languages which are denoted by regular tree expressions are proven to be equal, conversion can be performed. I focus on conversion from finite tree automata to regular tree expressions. During this thesis one state-of-the-art solution is presented and two new approaches are discovered and implemented. Brief comparison between algorithm and measurements are included.

Keywords regular tree expression, finite tree automaton, conversion

Contents

List of Figures

Introduction

In this thesis, I focus on regular *tree languages.* In a textual-language discipline there is a class of *regular languages* as well as its generator, *regular grammar*, its acceptor, *finite state automaton* (FA) and an alternative language descriptor *regular expression* (RE). Analogically in the world of computer science there exist regular tree language's class, *regular tree grammar*, *finite tree automaton* (FTA) and *regular tree expression* (RTE). Furthermore, the analogy between tree languages and textual languages is deeper. There exists deterministic and non-deterministic version of FTA, *determination* of non-deterministic FTA, *minimization* algorithm, *pumping lemma* for regular tree language [1], *Glushkow's algorithm* for conversion regular tree expressions to finite tree automata [9], equation method for conversion from FTA to RTE introduced in [10]. And this thesis is endeavour to find analogical algorithm for FA to RE conversion by state elimination method.

Motivation

The discipline of tree languages is not so widely explored such as textual-language discipline. However, it is widely used in computer science. Tree languages have been useful for a wide variety of problems such as code generation, indexing [5], cryptography [2], XML processing [8] and natural language processing [7].

Conversion from FTA to RTE can be performed by the mentioned equation method [10]. FTA can be determinized and minimized by exact algorithms. However, no such a minimization algorithm is known for RTE. A comparison between two RTEs is implemented by conversion of both to FTA followed by a comparison of their minimal forms. Therefore it is reasonable to figure out the algorithm, that gives as short and as intuitive results as possible. Moreover deeper examination of this analogy is topic important enough.

Goals

The primary goal is to design a new algorithm converting FTA to RTE that can be considered as an analogy to state elimination algorithm as was mentioned. The secondary goal is to design an algorithm that gives shorter or more natural results.

Achieved results

The first goal was achieved by the algorithm 4.1. However, this solution has weaknesses (see section 3.1.2). Source of sub-optimal constructions and confusions was denoted. Proposal for fixation was given and implemented. I proposed criterion on purpose to determine whether it helped. Measurements point out that the second solution is significantly better.

CHAPTER **1**

Used notation

In the beginning, definitions of tree language, finite tree automaton, regular tree expressions and some of the related terms will be useful to create foundations for this work. Used notation is based on [1]. However, the notation is simplified and extended on the purpose of this thesis. Let me firstly describe in general language followed by an exact formal definition.

1.1 Trees

I will use only rooted trees, where each element (ranked symbol) has ordered list of children. A tree is composed inductively by connecting other trees as children. And it ends by leaf, which is a symbol of zero ranks.

Definition 1.1.1 (Ranked symbol)**.** Let S is a symbol and $r \in \mathbb{N}; r \geq 0$ is a rank, then S_r is a ranked symbol. When rank is zero, it is so-call leaf. E.g. A_0 is a leaf.

Definition 1.1.2 (Ranked alphabet)**.** Let ranked alphabet is a couple ($\sum$, Arity), where $\sum$ is a finite set and Arity is a mapping from $\sum$ into $\mathbf{N}$. The arity (also known as rank) of symbol $h \in \sum$ is $Arity(f)$. Elements of arity 0 are called leaves or constants. Alphabet has to containing at least one leaf to be useful. Moreover let $\sum_i \subset \sum$ is a set of all symbols of arity i. E.g. $\sum_0$ is a set of leaves.

Remark 1.1.1. *When it is clear, the rank (arity) is usually in the index. Eg.* h_2 *is ranked symbol* h *with rank 2.*

Definition 1.1.3 (Extended alphabet by $\mathbb{X}$)**.** Let $\mathbb{X}$ be a set of leaves called substitution symbols. Substitutions symbols and alphabet are usually disjoint sets. This set is used as an extension for alphabet. Then $\sum \bigcup \mathbb{X}$ is extended alphabet.

Definition 1.1.4 (Tree)**.** Let ranked symbol $T \in \sum_r$ is the root of the tree with rank r, then follows r roots of nested trees e.g. $a_2(b_0, c_2(d_0, e_1(d_0)))$. Besides, let $L(\sum)$ is a set of all the trees over alphabet $\sum$.

I will be working with finite trees only, therefore I need at least one leaf in the alphabet.

Remark 1.1.2 (Tree representation)**.** *Trees can be either represented graphically or textually in strings. In my work I will use both representations, but mostly textual form.*

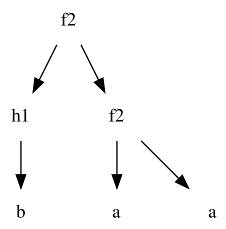


Figure 1.1: Example tree

For textual form I will use prefix notation, which is more common. Following tree is same as tree on the figure 1.1.2. $f_2(h_1(b_0), f_2(a_0, a_0))$

Definition 1.1.5 (Tree language)**.** Let $\sum$ be an alphabet, then $k \subset L(\sum)$ is a tree language over alphabet.

Definition 1.1.6 (Tree substitution, Language substitution)**.** Let $\sum \bigcup \mathbb{X}$ be an extended alphabet. Let $t, d_1, \ldots, d_n$ be trees over it and $\Box_1, \ldots, \Box_n \in \mathbb{X}$. Substitution is mapping from substitution symbols to trees. Substitution operation is operation that replaces substitution symbols by trees, so that result of substitution operation is original tree, but all substitution symbols covered by substitution are substituted by appropriate trees. Eg. Let $\sum = \{f_2, g_2, a, b\}$ and $\mathbb{X} = \{x_1, x_2\}$. Consider tree over extended alphabet $t = f(x_1, x_1, x_2)$ and following substitutions $s = \{x_1 \leftarrow a, x_2 \leftarrow g(b, b)\}$, $z = \{x_1 \leftarrow x_2, x_2 \leftarrow b\}$. Then $t \cdot s = f(a, a, g(b, b))$, $t \cdot z = f(x_2, x_2, b)$.

The language substitution is defined by following application of tree substitution. $L_1 \cdot sL_2 = \{t_1 \cdot t_2 | t_1 \in L_1, t_2, \in L_2\}$.

Definition 1.1.7 (Language iteration)**.** Let L be a language over extended alphabet and $\Box$ be substitution symbol from the same alphabet and n is number of iterations, then iteration $L^{n\square}$ is defined inductively as

$$\begin{aligned} L^{n\square} &= L \cdot \square E^{n-1\square} \\ L^{0\square} &= \square. \end{aligned} \tag{1.1}$$

Definition 1.1.8 (Iteration closure)**.** Let L be a language over extended alphabet and $\square$ be substitution symbol from the same alphabet Then closure $L^{\star\square}$ is defined as

$$L^{\star\square} = \bigcup_{k=0}^{\infty} L^{k\square}$$

1.2 Finite Tree Automaton

There are two types of tree automata one is top-down, which traverses a tree from its root, and another one is bottom up, which traverses a tree from leaves. In this work, I will use only bottom-up tree automata. Bottom-up types accept when the root of the tree is on the final state. Set of all the trees that automaton accepts is the language of the automaton. We consider two automata are equal when languages they are accepting are equal. These can be deterministic and non-deterministic, but we will prove their power equality.

Definition 1.2.1 (Finite tree automaton)**.** Finite tree automaton (FTA) over $\sum$ is a 4-tuple $A = (Q, \sum, F, \delta)$ where

$$\begin{aligned} Q &\quad \text{is a finite set of states,} \\ \sum &\quad \text{is an alphabet,} \\ F \subset Q &\quad \text{is a set of final states,} \\ \delta &\quad \text{is a set of transition.} \end{aligned} \tag{1.2}$$

Transition is of the following type:

$$f_r(q_1(x_1), \dots q_r(x_r)) \to q(f(x_1), \dots, f(x_r)),$$

where $r \geq 0, f_r \in \sum_r, q, q_1, \dots, q_r \in Q$, and $x_1, \dots, x_r$ are helper variables denoting trees.
FTA is always in class of non-deterministic FTA (NFTA), deterministic FTA (DFTA) is subset of NFTA. An tree automaton is DFTA, when there are no two transitions with same left-hand side in its set of transitions.

Remark 1.2.1 (Simplified transitions)**.** *Note that transitions defined above stores input in attributes of states and accumulates tree. Eg.: Consider following automaton.*

$$\begin{aligned} A = (\{A, F\}, &\{a_0, +_2\}, \{F\}, \delta) \\ \delta = \{ &+ (A(x_1), A(x_2)) \to F(+(x_1, x_2)), \\ &a \to A(a)\} \end{aligned} \tag{1.3}$$

Leaf a activates the second rule $a \to A(a)$ and two leaves a leading in $+$ activates the first rule $+(A(a), A(a)) \to F(+(a,a))$, then the automaton accepts input tree $+(a,a)$.

The property that automaton stores the input is appreciable, but it is not essential for this research. Therefore simplified notation can be used here.

Definition 1.2.2 (Transition in simplified notation)**.** Let $A = (Q, \sum, F, \delta)$ be FTA, then transition in simplified notation is

$$f_r(q_1, \dots q_r) \to q,$$

where $r \geq 0, f_r \in \sum_n, q, q_1, \dots, q_r \in Q$.

1.2.1 Notation of transition

For given DFTA (resp. NFTA) $A = (Q, \sum, F, \delta)$. Let me set specific names for parts of transition here. Let me break $t = (f_r(q_1, \dots q_r) \to s) \in \delta$ into ranked symbol $f_r \in \sum_r$ with rank r, let incoming states be a name of vector $(q_1, q_2, \dots, q_r)$, where $q_i \in Q$ for $i = 1, \dots, r$ and target state $s \in Q$.

The mapping which associates state to set of *incoming transitions to state* as well as mapping to *outgoing transition from state* can be stated, let call them TrTo and TrOut. Moreover let me set mapping that can be used to entry a part of transition. Let define *symbol of transition* as mapping transition to ranked symbol and call it Symb and *children* called Kids as incoming state into transition.

$$\begin{aligned}
&\mathrm{TrTo}(u) = \{(f_r(q_1, q_2, \dots q_r) \to s) \in \delta;\ \text{where } u = s\} \\
&\mathrm{TrFrom}(u) = \{(f_r(q_1, q_2, \dots q_r) \to s) \in \delta;\ \text{where } u \in q_1, \dots, q_r\} \\
&\mathrm{Symb}((f_r(q_1, q_2, \dots q_r) \to s)) = f_r \\
&\mathrm{Target}((f_r(q_1, q_2, \dots q_r) \to s)) = s \\
&\mathrm{Kids}((f_r(q_1, q_2, \dots q_r) \to s)) = (q_1, q_2, \dots q_r)
\end{aligned} \tag{1.4}$$

Following definition terms *language of state* and *language from edge* are defined inductively dependent each on other.

1.2.2 Language of automaton

Definition 1.2.3 (Language of state, language of edge)**.** Let $A = (Q, \sum, F, \delta)$ is a FTA. Let $s \in Q$ be a state, then define $L_{\text{state}}(s)$ as a language of state s. Which is composed from all the trees flowing into that state. These trees are flowing into state through transitions. Furthermore, language of transition is combination of languages from incoming children connected into one symbol, which is their common root.

$$\begin{aligned}
L_{\text{state}}(u) &= \bigcup_{t_k \in \text{TrTo}(u)} L_{\text{edge}}(t_k) \\
L_{\text{edge}}(t_k) &= \{f(x_1, \ldots, x_k) | x_1 \in L_{\text{state}}(c_1), \ldots, x_k \in L_{\text{state}}(c_k)\}, \\
&\quad \text{where } f = \text{Symb}(t), (c_1, \ldots, c_k) \in \text{Kids}(t_k)
\end{aligned} \tag{1.5}$$

Definition 1.2.4 (Language of automaton)**.** Let $A = (Q, \sum, F, \delta)$ is a FTA. Then $L(A)$ is a language composed of all trees acceptable by A.

$$L(A) = \bigcup_{f \in F} L(f)$$

1.2.3 Properties of tree automata

Definition 1.2.5 (Determinization algorithm)**.** Perform following steps on NFTA to obtain DFTA.

1. When more transitions of the same left-hand-side exists, then collect set of their targeting states. Create a new state representing that set of states.
2. Compose its name from sorted elements of set. For instance $\{C, A, B\} \rightarrow$ 'ABC' or $\{C, A\} \rightarrow$ 'AC'.
3. Replace group of mentioned redundant left-hand-side transitions by one transition leading to the new state.
4. Then collect all transitions leading from all of collected states and add similar transitions, but leading from the new state.
5. Continue step 1 until there is nothing to add.

Remark 1.2.2. *Determinization algorithm 1.2.5 is transcript from proof of Theorem 1.1.9 in [1].*

Lemma 1.2.3. *Algorithm will terminate and can't be worse then exponential in time.*

Proof. Let Q is the set of states of NFTA, it is finite set. Let $N = |Q|$ is a number of elements. This set has 2^N subsets, thus we can produce at most 2^N states in new DFTA. □

Theorem 1.2.4. *NFTA and DFTA are equally strong.*

Proof. Conversion exists in both directions.

1. Since by definition is DFTA is subset of NFTA, it is trivially fulfilled.
2. We can convert NFTA to DFTA using determinization 1.2.5.

Thus NFTA and DFTA are equally strong. □

1.3 Regular Tree Expression

This is another way to describe a tree language. Regular Tree Expression (RTE) is of a tree structure. It does consist ranked alphabet powered by special features. A tree can consist of alternation, iteration and substitution. It uses substitution symbols denoting where are trees connected and how they are generated.

Definition 1.3.1 (regular tree expressions)**.** The set TRegEx($\sum, \mathbb{X}$) of regular tree expressions over ranked alphabet $\sum$ extended by substitution symbols $\mathbb{X}$ is defined inductively:

$$\begin{aligned}
&\text{Empty set is in RTE.}\\
&E = \emptyset\\
&\text{Substitution symbol is in RTE.}\\
&E = \Box_x \quad \text{where } \Box_x \in \mathbb{X}\\
&\text{Alphabet ranked symbol followed by RTEs is also in RTE.}\\
&E = f_n(E_0, \dots, E_n) \quad \text{where } n \geq 0, f_n \in \sum_n \text{ and } E_i \in \text{TRegEx}(\textstyle\sum, \mathbb{X})\\
&\text{Alternation is in RTE}\\
&E = E_0 + E_1 \quad \text{where } E_i \text{ are RTEs}\\
&\text{Substitution by substitution symbol is RTE}\\
&E = E_0 \cdot \Box E_1 \quad \text{where } E_i \text{ are RTEs and } \Box_x \in \mathbb{X}.\\
&\text{Iteration by substitution symbol is RTE}\\
&E = E_0^{\star\Box} \quad \text{where } E_0 \text{ is RTE and } \Box_x \in \mathbb{X}.
\end{aligned} \tag{1.6}$$

Definition 1.3.2 (Language of RTE)**.** Let $E \in \text{TRegEx}(\sum, \mathbb{X})$, then $L(E)$ is a language composed by all the tree that E denotes. Language is defined by following equations

$$\begin{aligned}
&L(\emptyset) = \emptyset\\
&L(\Box_x) = \Box_x \quad \text{where } \Box_x \in \mathbb{X}\\
&L(f_n(E_0, \dots, E_n)) = \{f(x_1, \dots, x_n) | x_1 = L(E_0), \dots, x_n = L(E_n)\}\\
&L(E_0 + E_1) = L(E_0) \cup L(E_1)\\
&L(E_0 \cdot \Box E_1) = L(E_0) \cdot \{\Box \leftarrow L(E_1)\}\\
&L(E_0^{\star\Box}) = L(E_0)^{\star\Box}
\end{aligned} \tag{1.7}$$

1.3.1 Additional notation

Let me extend traditional notation of regular tree expressions by n-ary alternation.

Definition 1.3.3 (n-ary alternation)**.** Let $M = \{E_1, E_2, \dots, E_n\}$ be a set of RTEs, then n-ary alternation is defined as $E = \sum_{e \in M} e$ and its language is obviously $L(\sum_{e \in M} e) = \bigcup_{e \in M} L(e)$

Definition 1.3.4 (Contains)**.** Let $E \in \text{TRegEx}(\sum, \mathbb{X})$ and $\square \in \mathbb{X}$.

$$\begin{aligned} \text{contains}(E, \square) &= \mathit{true} \text{ when } e \text{ contains } \square \\ &= \mathit{false} \text{ otherwise} \end{aligned} \tag{1.8}$$

Remark 1.3.1. *Example RTE $h(a, \square)^{\star\square}$ is denoting following $\{\square, h(a_0, \square), h(a, h(a, \square)), \dots\}$. Note that this iteration always left substitution symbol after itself. Usually substitution symbols are not part of language's alphabet, therefore iteration is usually followed by substitution like $h(a, \square)^{\star\square} \cdot \square\, b$. Now it generates language $\{b,\ h(a, b),\ h(a, h(a, b)), \dots\}$.*

Here is an example iteration which, can terminate itself $h(a, b + \square)^{\star\square}$ generating $\{\square,\ h(a, b),\ h(a, \square), \dots\}$.

Lemma 1.3.2. *All iterations may leave substitution symbol after itself no matter how many loop it performed.*

Proof. Obviously from definition 1.1.7 for given number of iteration always exists case it left $\square$. $\square$

Theorem 1.3.3. *NFTA and RTE are equally strong.*

Proof. Conversion exists in both direction.

1. Follow this work to see conversion from NFTA to RTE. For instance method of equations 2.1 or dynamic programming approach 4.2.1.

2. Direction from RTE to NFTA is not included in this work, but follow master's thesis [9].

$\square$

Chapter 2

Observation about problem

Let me just introduce some related works and note what they came with. As the first contact with the topic I can mention the source [3], where is brief straight introduction to final tree automata and regular tree grammar.

One of the most relevant sources about tree languages is the book Tree Automata Techniques and Applications [1]. This book contains a wide range of definitions and properties, and shows connections with related topics. The book covers not just the basics on Tree Automata theory for finite ordered ranked trees, but also advanced applications and their variants such as constraining automata and tree transducers.

The last but not least is the diploma thesis Construction of a Pushdown Automaton Accepting a Language Given by Regular Tree Expression [9] written by Ing. Tomáš Pecka, who were inventing modification of Glushkov's Algorithm for conversion from RTE to Push-Down automata, which can be then converted to finite tree automata. It is a conversion in opposite direction, than I do.

Remark 2.0.1 (Relation between finite tree automata and pushdown automata)**.** *Regular tree languages are closely related with context-free textual languages. As it was mentioned (remark 1.1.2), tree can be represented textually. It can be proved that regular tree grammar is special case of context-free textual grammar [4]. Thus pushdown automaton can accept textual form of regular tree language. Pushdown automata of this type were able to be converted to equivalent finite tree automata. However it is not true that any pushdown automata can be converted to FTA, because regular tree grammar is weaker then context-free textual grammar [6].*

Existing solutions

There is a constructive proof of Kleene's Theorem for Tree Languages in the source [1]. This theorem is based there on preposition that says "for any finite tree automaton, there exists regular tree expression denoting equal language".

There is a paper [10] introducing solution with equations. To cut long story short they simply creates equation system from transition function of FTA. Then they solve equation system by substituting variables until they can construct a result. This solution is described in following section in my notation.

2.1 Solution with equations

For given automaton $A = (Q, \sum, F, \delta)$ let's create system of equations first. The system of equations is based on TRegEx($\sum, \mathbb{X}$). Where $\mathbb{X}$ is subset of $\{X_s | s \in Q\} \cup \{x_s | s \in Q\}$ Each state s is considered as variable X_s, therefore is present in $\mathbb{X}$ and when iteration is necessarily substitution symbol x_s appears. The variable is equal to summation of symbols leading the state. Those symbols are connected to appropriate variables of states.

Definition 2.1.1 (equation of state)**.** Let $s \in Q$ be state and $T = \mathrm{TrTo}(s)$ incoming transitions. Then equation of state is defined as this.

$$X_s = \sum_{t \in T} \mathrm{Symb}(t)\mathrm{XKids}(t)$$

Where X_s is variable of state s, XKids(t) is similar to Kids(t) but it returns vector of state's variables instead of vector of states. Eg.: $\mathrm{Kids}(t) = (2,2,1) \Rightarrow \mathrm{XKids}(t) = (X_2, X_2, X_1)$

See on the example.

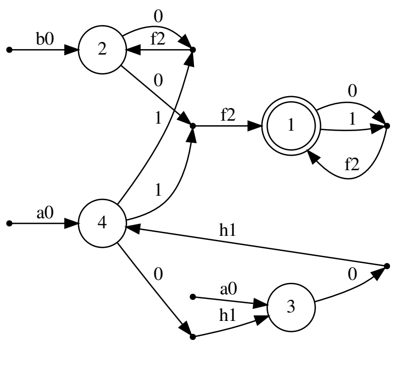


Figure 2.1: Example automaton

For instance there into state 1 are leading exactly two different transitions of symbol f_2. Therefore for first state it is $X_1 = f_2(X_1, X_1) + f_2(X_2, X_4)$, and ranks seam to look redundant here, so just omit them. I will generate whole system in this manner.

$$\begin{aligned} X_1 &= f(X_1, X_1) + f(X_2, X_4) \\ X_2 &= b + f(X_2, X_4) \\ X_3 &= a + h(X_4) \\ X_4 &= a + h(X_3) \end{aligned} \tag{2.1}$$

2.1.1 Solving system of equations

According to [10] article, we can solve the obviously by substituting, when definition of state is equation dependent on itself, then it is a loop, which can be solved by iteration. Performing of substitution $X_3 \to X_4$ makes a loop.

$$\begin{aligned} X_1 &= f(X_1, X_1) + f(X_2, X_4) \\ X_2 &= b + f(X_2, X_4) \\ X_3 &= a + h(X_4) \\ X_4 &= a + h(a + h(X_4)) \quad \leftarrow \text{loop here} \end{aligned} \tag{2.2}$$

Of course substitution here does not help, but let's expand it few times for observation.

$$X_4 = a + h(a + h(a + h(a + h(a + h(a + h(X_4))))))$$

After peruse of expression, note that X_4 is composed out of non-loop parts and loop parts. Non-loop part surely terminates a loop.

$$X_4 = \underbrace{a}_{\text{non-loop}} + \underbrace{h(a + h(X_4))}_{\text{loop}}$$

RTE iteration can be straightforwardly implemented from the loop as $h(a + h(x_4))^{\star x_4}$ by using substitution symbol x_4. It can either leave substitution symbol or terminate itself by $h(a)$. When it leaves x_4 symbol it should be substituted by any of non-loop parts of X_4. Thus final RTE of state 4 has to be $X_4 = h(a + h(x_4))^{\star x_4} \cdot x_4\, a$.

Definition 2.1.2 (Loop and non-loop parts)**.** Let X_s be a variable denoting state $s \in Q$ and R_s be a set of all options defining $X_s = \sum_{r \in R_s} r$, so that equality holds. There is no other variable then X_s on the right side. Then define loop and non-loop parts as

$$\begin{aligned} \text{loop}(X_s) &= \{r \in R_s; \text{contains}(r, X_s)\} \\ \text{non-loop}(X_s) &= \{r \in R_s; \neg\text{contains}(r, X_s)\} \end{aligned} \tag{2.3}$$

Theorem 2.1.1. *When equation with variable X_s has on the right hand side just variables X_s and no other variable. It can be always handled as this.*

$$X_s = \left(\sum_{o \in loop(X_s)} o \right)^{\star X_s} \cdot X_s \left(\sum_{m \in non\text{-}loop(X_s)} m \right)$$

Proof. I am starting with equation from definition 2.1.2

$$\begin{aligned}
X_s &= \sum_{r \in R_s} r \\
X_s &= \sum_{o \in \text{loop}(X_s)} o + \sum_{m \in \text{non-loop}(X_s)} m \\
X_s &= \left(\sum_{o \in \text{loop}(X_s)} o + \sum_{m \in \text{non-loop}(X_s)} m \right)^{\star X_s} \cdot X_s \emptyset \\
&\text{where } \emptyset \text{ is empty symbol}
\end{aligned} \tag{2.4}$$

Loop can be terminated by either loop alternative if there is that option or by non-loop variant. When non-loop variant is chosen it terminates a loop surely. Thus non-loop part can be used as a substitute for X_s symbol after loop.

$$X_s = \left(\sum_{o \in \text{loop}(X_s)} o \right)^{\star X_s} \cdot X_s \left(\sum_{m \in \text{non-loop}(X_s)} m \right) \tag{2.5}$$

□

$$\begin{aligned}
X_1 &= f(X_1, X_1) + f(X_2, X_4) \\
X_2 &= b + f(X_2, X_4) \\
X_3 &= a + h(h(a + h(x_4))^{\star x_4} \cdot x_4 \ a) \\
X_4 &= h(a + h(x_4))^{\star x_4} \cdot x_4 \ a
\end{aligned} \tag{2.6}$$

$$\begin{aligned}
X_1 &= f(X_1, X_1) + f(X_2, X_4) \\
X_2 &= b + f(X_2, h(a + h(x_4))^{\star x_4} \cdot x_4 \ a) \\
X_3 &= a + h(h(a + h(x_4))^{\star x_4} \cdot x_4 \ a) \\
X_4 &= h(a + h(x_4))^{\star x_4} \cdot x_4 \ a
\end{aligned} \tag{2.7}$$

$$\begin{aligned}
X_1 &= f(X_1, X_1) + f(X_2, X_4) \\
X_2 &= f(x_2, h(a + h(x_4))^{\star x_4} \cdot x_4 \ a)^{\star x_2} \cdot x_2 \ b \\
X_3 &= a + h(h(a + h(x_4))^{\star x_4} \cdot x_4 \ a) \\
X_4 &= h(a + h(x_4))^{\star x_4} \cdot x_4 \ a
\end{aligned} \tag{2.8}$$

$$\begin{aligned}
X_1 &= f(X_1, X_1) + f(f(x_2, h(a + h(x_4))^{\star x_4} \cdot x_4 a)^{\star x_2} \cdot x_2 b, h(a + h(x_4))^{\star x_4} \cdot x_4 a) \\
X_2 &= f(x_2, h(a + h(x_4))^{\star x_4} \cdot x_4 \ a)^{\star x_2} \cdot x_2 \ b \\
X_3 &= a + h(h(a + h(x_4))^{\star x_4} \cdot x_4 \ a) \\
X_4 &= h(a + h(x_4))^{\star x_4} \cdot x_4 \ a
\end{aligned} \tag{2.9}$$

Finally there is the solution of equation system.

$$\begin{aligned}
&X_1 = f(x_1, x_1)^{\star x_1} \cdot x_1 f(f(x_2, h(a + h(x_4))^{\star x_4} \cdot x_4 a)^{\star x_2} \cdot x_2 b, h(a + h(x_4))^{\star x_4} \cdot x_4 a)\\
&X_2 = f(x_2, h(a + h(x_4))^{\star x_4} \cdot x_4\ a)^{\star x_2} \cdot x_2\ b\\
&X_3 = a + h(h(a + h(x_4))^{\star x_4} \cdot x_4\ a)\\
&X_4 = h(a + h(x_4))^{\star x_4} \cdot x_4\ a
\end{aligned} \tag{2.10}$$

Result is summation of what is on final states, in this case final is just 1. Thus X_1 is a solution.

Remark 2.1.2. *Order of substitution has effect to the result. For example when I would substitute in X_3 first, then I am obtaining loop on X_3 instead of X_4 and this change leads to different result later.*

$$\begin{aligned}
X_1 &= f(X_1, X_1) + f(X_2, X_4)\\
X_2 &= b + f(X_2, X_4)\\
X_3 &= h(a + h(x_3))^{\star x_3} \cdot a\\
X_4 &= a + h(X_3)
\end{aligned} \tag{2.11}$$

Chapter 3

New approach

3.1 State elimination approach

The algorithm presented in this section is similar to constructive proof of preposition 2.2.7. from source [1]. The preposition is there used to prove Kleene's Theorem for Tree Languages.

The first what I was asked to do was state elimination approach. Let me take into consideration whether a state of given FTA can be substituted by RTE. Note that each state of the automaton denotes its language. Each RTE also denotes the language. Let me suppose that an RTE that denotes the same language as the state can be constructed. Then all outgoing transitions from the state that was exchanged are kept in the automaton, child of that state is replaced by appropriate RTE.

The automaton seams to be broken, since it is not a valid automaton according to definition. Nevertheless, it will be proved, that this construct is representing the same language as original automaton. Following lemma (3.1.1) describes how it could be achieved.

Definition 3.1.1 (Eliminated states)**.** Let $A = (Q, \sum, F, \delta)$ be an automaton with Q set of states, then $\overline{Q}$ is a set of eliminated states. This set is initially empty for every valid automaton. However, it will be filled during algorithm.

Definition 3.1.2. (Links) Let q be a RTE. Then define

$$\text{Links}(q) = \{\text{each substitution symbol in } q\}$$

Definition 3.1.3 (self-loop and out-to-in transition)**.** Let $A = (Q, \sum, F, \delta)$ be an automaton and $s \in Q$ be a state, then all its incoming transitions are either self-loop or out-to-in.

$$\exists t = (f_r(q_1, q_2, \dots q_r) \to s) \in \mathrm{TrTo}(s)$$
$$t \begin{cases} \text{is self-loop transition} & \text{if } s \in \mathrm{Kids}(t) \text{ or } \exists q \in \mathrm{Kids}(t) \land q \in \overline{Q} \land s \in \mathrm{Links}(q) \\ \text{is out-to-in transition} & \text{otherwise} \end{cases} \tag{3.1}$$

Let me admit for following algorithm that state can be interchanged by RTE and RTE can use states of automaton as substitution symbol.

Definition 3.1.4 (State extends RTE)**.** Let $A = (Q, \sum, F, \delta)$ be an automaton and $s \in Q$ be its state. Then s is also special substitution symbol generating $L(s)$ and can be substituted.

Lemma 3.1.1 (State substitution)**.** *Let $A = (Q, \sum, F, \delta)$ be a FTA and $s \in Q; s \notin \overline{Q}$ be its state. It is possible to substitute a state s by RTE to get new FTA A' with $\overline{Q'} = \overline{Q} \bigcup \{s\}$ so that equations holds $L(A) = L(A')$.*

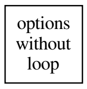
Proof. Language of state is defined

$$L(s) = \bigcup_{t_k \in \mathrm{TrTo}(s)} L_{\mathrm{edge}}(t_k) = L(s)$$

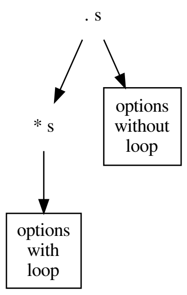
, equal RTE can be obtained by following construction.

$$\begin{aligned} O &= \{t \in \mathrm{TrTo}(s); t \text{ is self-loop}\} \\ M &= \{t \in \mathrm{TrTo}(s); t \text{ is out-to-in}\} \\ t &= \begin{cases} \text{when } s \text{ has any self-loop} \\ (\sum_{o \in O} \mathrm{Symb}(o)(\mathrm{Kids}(o)))^{\star s} \cdot s((\sum_{m \in M} \mathrm{Symb}(m)(\mathrm{Kids}(m)))) \\ \text{otherwise} \\ (\sum_{m \in M} \mathrm{Symb}(m)(\mathrm{Kids}(m))) \end{cases} \end{aligned} \tag{3.2}$$

Graphical explanation may suit better here. We can simply create RTE from transition by transition's ranked symbol connected to appropriate children.



To create RTE from state, there are two options. If state has no self-loop transition, it is just alternation from out-to-in transition's RTEs. It is obviously right.



When any self-loop exists, then self-loop's RTEs are to be under iteration and out-to-in's RTEs will be under substitution. Here it could be anything under iteration out-to-in options as well as self-loop options. In that case out-to-in option always terminates the iteration. Thus there is no need for return into the loop. Because of substitution symbol may be left after any number of iterations (lemma 1.3.2), therefore it can be out of loop under substitution.

Final language of automaton (according to definition 1.2.4) is dependent on language of state (definition 1.2.3) When state $s \in \overline{Q}$ to obtain its language RTE will be used.

$$L(s) = \begin{cases} L_{\text{state}}(s) & \text{if } s \notin \overline{Q} \\ L_{\text{RTE}}(s) & \text{if } s \in \overline{Q} \end{cases} \tag{3.3}$$

Language of edge need just a minor change.

$$\begin{aligned} L_{\text{edge}}(t_k) = \{f(x_1, \ldots, x_k) | x_1 \in L(c_1), \ldots, x_k \in L(c_k)\}, \\ \text{where } f = \text{Symb}(t), (c_1, \ldots, c_k) \in \text{Kids}(t_k) \end{aligned} \tag{3.4}$$

Then language of automaton is not affected at all. □

Corollary 3.1.2. *Simple algorithm can be done.*

Definition 3.1.5 (State Elimination Algorithm)**.** An algorithm performs conversion from FTA to RTE.
Input: NFTA $A = (Q, \sum, F, \delta)$, Output: equivalent RTE

1. Initialisation. Create new NFTA $A' = (\sum', Q', \delta', F')$ that has new state Y; $Q' = Q \bigcup \{Y\}$. This new state is only one final $F' = \{Y\}$ and all of finals of original automaton are leading into Y through special unary symbol ϑ transition $\delta' = \delta \bigcup \{(q, \vartheta) \to \{Y\} : q \in F\}$.

2. $\forall u \in Q$ do

   Create a language definition of state u in RTE by following rules:

a) If u has a self-loop, then all its incoming loop transitions sum under iteration. Careful when transition is leading from RTE, check all its substitution symbol, transition can be self-loop by backward substitution symbol. Use state symbol as substitution symbol. Then substitute symbol by summation of all out-to-in transitions. Otherwise just combine out-to-in transitions.

b) Remove incoming transitions from automaton, and replace state by expression

3. Collect RTEs from transitions leading to Y, cut off their roots (unary symbols ϑ) and sum them up. It is a result.

Example

Let me apply an algorithm on following example.

$$
\begin{aligned}
A &= (\textstyle\sum, Q, \delta, F) \\
\textstyle\sum &= \{0_0, 1_0, \mathrm{or}_2, \mathrm{and}_2\} \\
Q &= \{t, f\} \\
\delta &= \{ \\
&\quad 1 \to \{t\}, \\
&\quad 0 \to \{f\}, \\
&\quad \mathrm{and}_2(t,t) \to \{t\}, \\
&\quad \mathrm{and}_2(f,t) \to \{f\}, \\
&\quad \mathrm{and}_2(t,f) \to \{f\}, \\
&\quad \mathrm{and}_2(f,f) \to \{f\}, \\
&\quad \mathrm{or}_2(t,t) \to \{t\}, \\
&\quad \mathrm{or}_2(f,t) \to \{t\}, \\
&\quad \mathrm{or}_2(t,f) \to \{t\}, \\
&\quad \mathrm{or}_2(f,f) \to \{f\} \\
\}
\end{aligned}
\tag{3.5}
$$

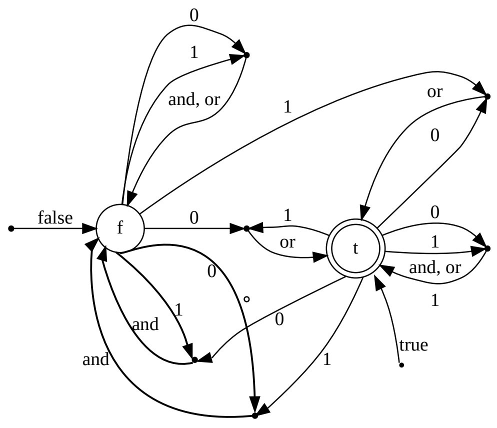


Figure 3.1: Diagram of original automaton.

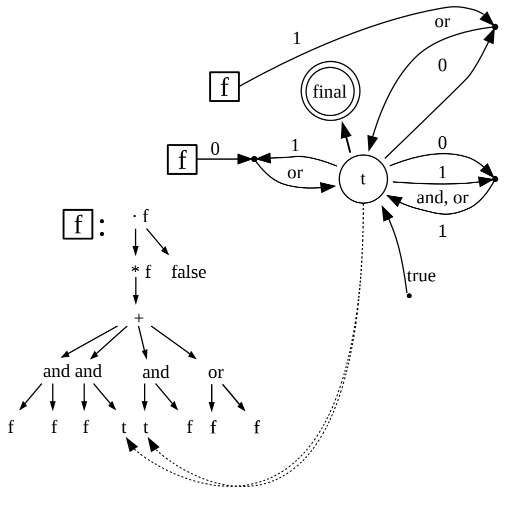


Figure 3.2: State f is eliminated.

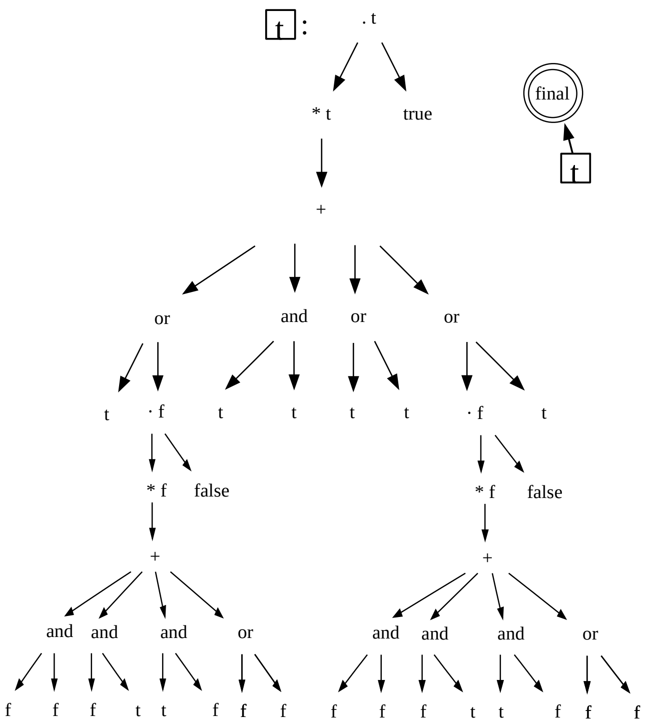


Figure 3.3: State t is eliminated, finish.

This algorithm is trivial and is analogical to state elimination for conversion from FSM to RE. The algorithm in this variant is not implemented during my research. I rather focused on more practical versions. Two sources that causes sub-optimal results are described bellow. The first one is redundancies (3.1.1) and second one is order sensitivity (3.1.2).

3.1.1 Redundancies

The result tree may contain a lot redundant sub-trees. Some redundancies can be trivially resolved. For example two same children under symbol could be optimised by substitution above symbol. Another redundancies are not easy to reduce.

3.1.2 Order does matter

Performing elimination in different order gives a different result. This method is very order sensitive. Some of orders are better than others. Consider previous example. Elimination was performed in order t, f and nice result was obtained.

Result when f is first

$$\begin{aligned}&(\mathit{and}(t,t)\\&+\mathit{or}(t,(\mathit{and}(f,f)+\mathit{and}(f,t)+\mathit{and}(f,t))+\mathit{or}(f,f))^{\star f}\cdot f(\mathit{false}))\\&+\mathit{or}((\mathit{and}(f,f)+\mathit{and}(f,t)+\mathit{and}(f,t))+\mathit{or}(f,f))^{\star f}\cdot f(\mathit{false}),t)\\&+\mathit{or}(t,t))^{\star t}\cdot t(\mathit{true})\end{aligned} \tag{3.6}$$

But starting with state t.

Reaching from t state

$$(\mathit{and}(t,t)+\mathit{or}(t,t)+\mathit{or}(t,f_{\text{state}})+\mathit{or}(f_{\text{state}},t))^{\star t}\cdot t(\mathit{true})$$

. Note there is no loop through f. Since t is eliminated, it is just a RTE on incoming edge and there is an existence of self-loop for f leading through RTE. After elimination of f getting

$$\begin{aligned}&(\mathit{and}(f,f)\\&+\mathit{and}(f,(\mathit{and}(t,t)+\mathit{or}(t,t)+\mathit{or}(t,f)+\mathit{or}(f,t))^{\star t}\cdot t(\mathit{true}))\\&+\mathit{and}(f,(\mathit{and}(t,t)+\mathit{or}(t,t)+\mathit{or}(t,f)+\mathit{or}(f,t))^{\star t}\cdot t(\mathit{true}))\\&+\mathit{or}(f,f))^{\star f}\cdot f(\mathit{false})\end{aligned} \tag{3.7}$$

Situation is that f is connected to Y through RTE. Whole problem is that this is just a RTE of state f, however, t is leading into final state.

Result when t is first

$$\begin{aligned}&(\mathit{and}(t,t)\\&+\mathit{or}(t,t)\\&+\mathit{or}(t,\\&\quad(\mathit{and}(f,f)\\&\quad+\mathit{and}(f,(\mathit{and}(t,t)+\mathit{or}(t,t)+\mathit{or}(t,f)+\mathit{or}(f,t))^{\star t}\cdot t(\mathit{true}))\\&\quad+\mathit{and}(f,(\mathit{and}(t,t)+\mathit{or}(t,t)+\mathit{or}(t,f)+\mathit{or}(f,t))^{\star t}\cdot t(\mathit{true}))\\&\quad+\mathit{or}(f,f))^{\star f}\cdot f(\mathit{false}))\\&+\mathit{or}(\\&\quad(\mathit{and}(f,f)\\&\quad+\mathit{and}(f,(\mathit{and}(t,t)+\mathit{or}(t,t)+\mathit{or}(t,f)+\mathit{or}(f,t))^{\star t}\cdot t(\mathit{true}))\\&\quad+\mathit{and}(f,(\mathit{and}(t,t)+\mathit{or}(t,t)+\mathit{or}(t,f)+\mathit{or}(f,t))^{\star t}\cdot t(\mathit{true}))\\&\quad+\mathit{or}(f,f))^{\star f}\cdot f(\mathit{false})\\&,t))^{\star t}\cdot t(\mathit{true})\end{aligned} \tag{3.8}$$

3.1.2.1 What causes such a differences?

Basically algorithm is doing a tree structure from a graph structure. Cycles are represented as RTE's iterations. It works, however result can be counter-intuitive.

Lemma 3.1.3. *Every cycle in given FTA will be once processed as self-loop transition during state elimination process. And RTE iteration of that cycle wins the last eliminated state from that cycle.*

Proof. Let $s_1, s_2, \dots, s_n$ are states of some FTA forming a cycle

$$s_1 \to s_2 \to \dots s_n \to s_1$$

When s_k is eliminated, there state $s_{(k+n-1) \bmod n}$ is leading to RTE of s_k and RTE of s_k is leading to state $s_{(k+1) \bmod n}$.

$$\dots \to s_{(k+n-1) \bmod n} \to \mathrm{RTE} s_k \to s_{(k+1) \bmod n} \to \dots$$

This principal continues until state s_h is the last and all others are eliminated.

$$\dots \to \mathrm{RTE} s_{(h+n-1) \bmod n} \to s_h \to \mathrm{RTE} s_{(h+1) \bmod n} \to \dots$$

Cycle is now self-loop and gained iteration is on RTEs_h. □

Lemma 3.1.4. *Let $s_1, s_2, \dots, s_n$ are states of some FTA forming a cycle, s_h is an iteration winner and Y final state. When path from s_h to final state leads through $s_1 \neq s_h$ another state of cycle. Then cycle fragment $s_1 \rightsquigarrow s_h$ has its RTE above the iteration and then again inside the loop.*

Proof.

$$s_1 \to s_2 \to \dots \to s_h \to \dots s_n \to s_1 \rightsquigarrow Y$$

Algorithm generates following result.

$$\begin{tikzcd}
& Y \arrow[d, squiggly] & \\
& s_1 \arrow[d, squiggly, "\text{from } s_1 \text{ to } s_h"] & \\
& \cdot s_h \arrow[dl] \arrow[dr] & \\
\star s_h \arrow[d, squiggly] & & \text{out-to-ins} \\
s_1 \arrow[d, squiggly, "\text{from } s_1 \text{ to } s_h"] & & \\
s_h & &
\end{tikzcd}$$

Fragment from $s_1 \rightsquigarrow s_h$ is there twice. The iteration is not situated well here. □

Corollary 3.1.5. *When s_1 is winner, there is not such a segment.*

$$\begin{tikzcd}
& Y \arrow[d, squiggly] & \\
& \cdot s_1 \arrow[dl] \arrow[dr] & \\
\star s_1 \arrow[d, squiggly] & & \text{out-to-ins} \\
s_n & &
\end{tikzcd}$$

Definition 3.1.6 (Well-situated iteration)**.** Let $s_1, s_2, \ldots, s_n$ are states of some FTA forming a cycle, s_h is an iteration winner, Y is final state and path from Y to s_h does not lead through another state of cycles. Thus there is not redundant path above integration and under iteration (as in Lemma 3.1.4). Then I call this well-situated iteration.

Path redundancies not only make results much longer but also less understandable. To reach maximum advantages from this lemma 3.1.4 I need to figure out an algorithm based on a different principle, which is focusing the next chapter.

Chapter 4

Enhanced solutions

In this chapter, I focus on optimisation of naive solution. In previous chapter sources of sub-optimal results have been described. I am introducing proposal of improvement for each issue. However, I have not figured out how I could combine these solutions to optimise both simultaneously.

4.1 Removal of State Approach – Optimisation by Substitution

This approach aims to solve the redundancy problem 3.1.1. The main idea is in different construction strategy. The algorithm starts with the sum of substitution symbols each for one of the final states. This particular result requires substitution above. Thus these are inserted in the stack. States on the stack are to be eliminated. Each state is eliminated at most once. When node from the stack is eliminated, every transition is handled as a symbol with substitution symbols denoting incoming states. These children could be of 3 types. It can be the state itself, then it indicates trivial self-loop. In the second case, it is state which has not been eliminated yet. This state requires substitution above, thus it is pushed to the stack. Moreover for that state of elimination is this child stored in the set and this set is accessible by dictionary through the state of elimination (I call it *leadingUp*). The last type of child is a state which is already eliminated. This state is already solved in a particular result, but it occurred again. Thus this state has to be resolved again with all its *leadingUp*s that are already eliminated and inductively *leadingUp*s that are eliminated of *leadingUp*s that are eliminated. This sub-problem resolves function *ComposeClosure*. Transitions are separated into two classes self-loops and out-to-ins. Loop can lead through eliminated states, the function *IsSelfLoop* is responsible for this traversal. After each elimination particular result grows by new substitution. When all the states are substituted, the result is complete.

Algorithm 1 Removal of state approach

```
Input: NFTA $A = (Q, \sum, F, \delta)$.
Output: an equivalent RTE.
 1: procedure ToRTE($A = (Q, \sum, F, \delta)$)
 2:     $S \leftarrow$ new Stack()
 3:     $R \leftarrow \emptyset$
 4:     for all $f \in F$ do
 5:         $S$.push($f$)
 6:         $R \leftarrow R \bigcup \{$SubstSymb($f$)$\}$
 7:     leadingToMe $\leftarrow$ new Dictionary¡State, Set¡State»()
 8:     leadingUp $\leftarrow$ new Dictionary¡State, Set¡State»()
 9:     for all $s \in Q$ do
10:         for all $t \in$ TrTo($s$) do
11:             for all neighbour $\in$ Kids($t$) do
12:                 leadingToMe[$s$].Insert(neighbour)
13:     Result $\leftarrow \sum R$
14:     ToRTE $\leftarrow$ new Dictionary()
15:     covered $\leftarrow$ new Set()
16:     ...                                    ▷ Procedure continues on the next page.
```

Algorithm 2 Removal of state approach — continue

```
...                                        ▷ Procedure begins on the previous page.
while $S$.NotEmpty() do
    current ← $S$.pop()
    if current ∈ covered then
        continue
    loopEdges ← ∅, outToInEdges ← ∅
    for all $t$ ∈ TrTo(current) do
        edgeRTE ← new RTE()
        edgeRTE.setSymbol(Symbol($t$))
        doesLinkToCurrent ← false
        statesToSubstitute ← new Set()
        for all neighbour ∈ Kids($t$) do
            if neighbour ∈ covered then
                statesToSubstitute.Insert(neighbour)
            else if neighbour ≠ current then
                mentioned.Push(neighbour)
                leadingUp[current].Insert(neighbour)
            if IsSelftLoop(covered, leadingToMe, neighbour, current) then
                doesLinkToCurrent ← true
            edgeRTE.addKid(Symbol(neighbour))
        for all n ∈ statesToSubstitute do
            with ← ComposeClosure(covered, leadingUp, stateToRte, n)
            edgeRTE ← Substitution(edgeRTE, with, Symbol(n))
        if doesLinkToCurrent then
            loopEdges.insert(edgeRTE)
        else
            outToInEdges.insert(edgeRTE)
    outToIn ← $\sum$ outToInEdges
    currentRTE ← outToIn
    if loopEdges.any() then
        loopRTE ← Iteration($\sum$ loopEdges, Symbol(current))
        currentRTE ← Substitution(loopRTE, outToIn, Symbol(current))
    stateToRte[current] ← currentRTE
    result ← Substitution(result, currentRTE, Symbol(current))
    covered.Insert(current)
return result
```

Algorithm 3 IsSelftLoop

Input: covered, leadingToMe, *from*(neighbour), *to*(current) and visited $\leftarrow$ new Dictionary()
Output: True if *from* leads back to *to*.

```
1: procedure IsSelfLoop(covered, leadingToMe, from, to, visited)
2:     if from = to then return true
3:     if from $\notin$ covered then return false
4:     if visited[from] then return false
5:     visited[from] $\leftarrow$ true
6:     result $\leftarrow$ false
7:     for all state $\in$ leadingToMe[from] do
8:         if isSelfLoop(covered, leadingToMe, state, to, visited) then
9:             result $\leftarrow$ true
10:    return result
```

Algorithm 4 ComposeClosure

Input: covered, leadingUp, stateToRte, state
Output: RTE that denotes state.

```
1: procedure ComposeClosure(covered, leadingUp, stateToRte, state)
2:     toSubs $\leftarrow$ new Vector()
3:     stack $\leftarrow$ new Stack()
4:     stack.Push(state)
5:     while stack.NotEmpty() do
6:         current $\leftarrow$ stack.pop()
7:         if current $\notin$ covered $\vee$ current $\in$ toSubs  then continue
8:         toSubs.PushBack(current)
9:         for all other $\in$ leadingUp[current do
10:            if other $\in$ covered then
11:                stack.Push(other)
12:    toSubs.popFront()
13:    result $\leftarrow$ stateToRte[state]
14:    for all other $\in$ toSubs do
15:        rte $\leftarrow$ stateToRte[other]
16:        result $\leftarrow$ Substitution(result, rte, Symbol(other))
17:    return result
```

Routine *ComposeClosure* may be considered as interesting. The dictionary *stateToRte* stores expression of the state s from the moment it was eliminated. The set of states directly reachable from that state s that were eliminated before state s has to be resolved inside its RTE. States that are reachable from state s (its *leadingUp*s) that are not eliminated are expected to be eliminated later, and substituted above. So only what has to be resolved here is a set of states that are its eliminated of its *leadingUp*s and state that are reachable through eliminated *leadingUp* of its eliminated *leadingUp*s and inductively. This composes closure, which consist of at most all eliminated states.

4.1.1 Time complexity

Let me analyse algorithm's time complexity for input automaton $A = (Q, \sum, F, \delta)$ Let T is number of outgoing connections in the automaton. This can be evaluated as accumulated rank over all transitions.

$$T = \sum_{h \in S} \text{Rank}(h),$$

where S is list of all symbols, for each transition one symbol.

Algortithm is base on DFS traversal and inside are called subroutines IsSelfLoop and ComputeClosure both are base on DFS. Thus final time complexity is $O(S + T(S + T))$.

4.2 Dynamic programming approach

In the previous chapter, I discussed the state elimination algorithm, and its disadvantage demonstrated on the example. In this chapter, I will endeavour to avoid redundancies of the type introduced by lemma 3.1.4. At first, I have to mull over whether the change of elimination order preventing the problem. The following example 4.2.1 points towards the fact that it is not just about the order.

4.2.1 Example cyclic fragment

Consider following fragment of automaton with cycle. There is just one cycle, but more are final states. State elimination approach permits just one winner, although this is not the only way how it can be done. Loop can be processed for each access point separately.

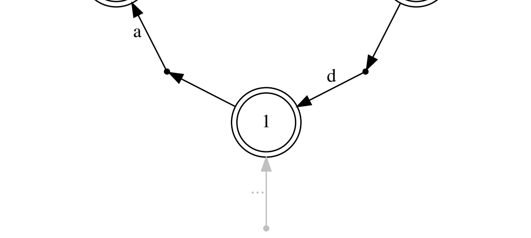


Figure 4.1: Fragment of FTA.

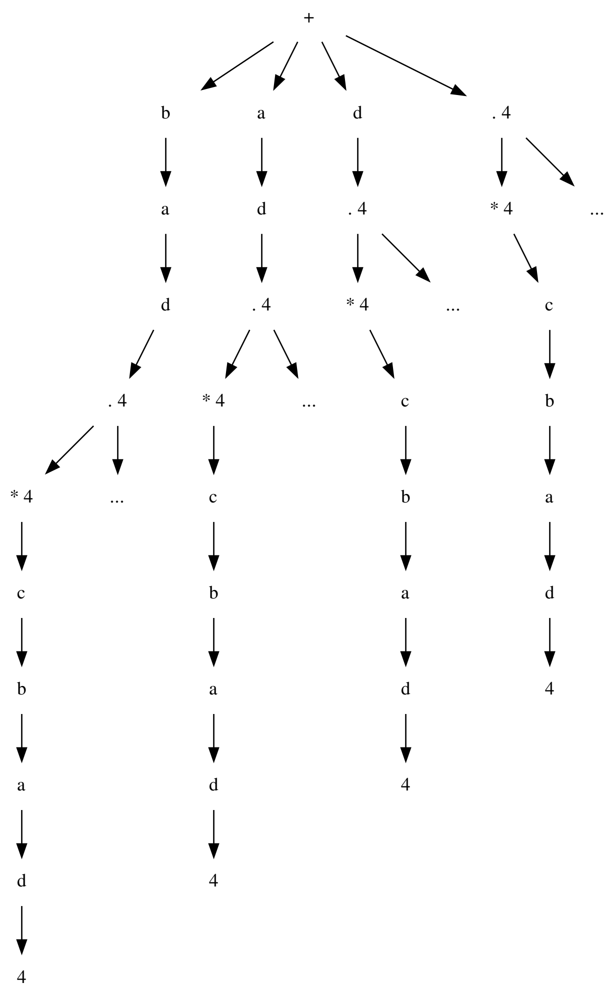


Figure 4.2: Fragment of FTA converted by state elimination.

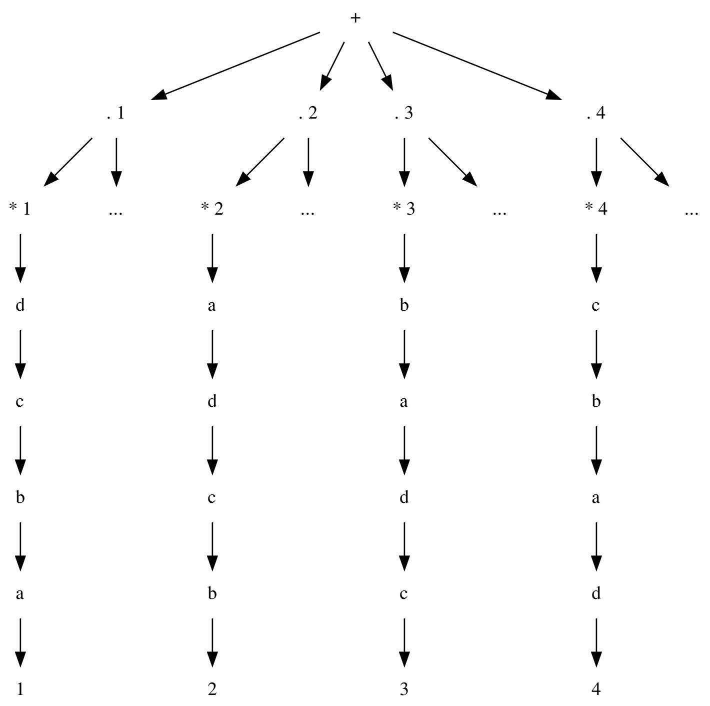


Figure 4.3: Possible optimisation of iterations.

However, the presented approach renders substitution optimisation impossible, because the common part is lost.

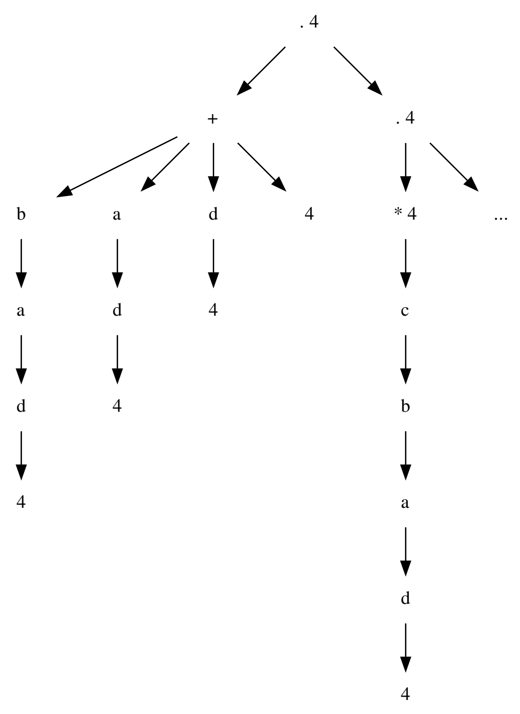


Figure 4.4: Possible optimisation by substitution.

4.2.2 Approach against the flow

Let me remind what is flowing into the state is a language of that state (Definition 1.2.3). FTA's language is the union of languages over all finite states (Definition 1.2.4). Let me deliberate construction of RTE during graph traversal from the final state against the flow. To complete RTE of a state, it is necessary to have RTEs of incoming transition. Incoming transitions are naturally children of the state, because of traversal against the flow. Thus recursion seems to be suitable for this. Consider recursive Depth-First Search or recursive Breadth-First Search as a base of the algorithm. The RTE of the state can be compiled in post order. Furthermore, it is possible to detect a cycle.

Usually, traversal generates a sequence of visited nodes, and all nodes are visited at most once. Nevertheless, this traversal will be entirely different in this way.

Consider an automaton with cycle and two different of its states are connecting the cycle to final. Then in the result, the cycle has to be processed twice. Firstly the winner is the first-mentioned state. Secondly, the winner is the second mentioned state. Naturally, the question comes to the mind "What is the termination condition and does algorithm even terminate?"

Recursion generates a call tree. On the stack is always the path to the initial call, which is here also the node's flow to the final, because the call was initiated from the final state against the flow.

Because of this, more corollaries can be stated. When child if some edge is on the stack path, it has to be a cycle. Moreover, that state is a great candidate to initiate an iteration, because the loop will be well-situated (proof 4.2.2). The termination condition is defined as the current state is not on stack path instead of the current state is not visited as ordinarily. It can be proven, that algorithm terminates, from fact, that stack path consist sequence of unique states (according to termination condition). And number of state state is finite (definition of FTA 1.2.1). Therefore there is at most $|Q|!$ of stack paths.

Theorem 4.2.1. *A transition leading to a state from stack path indicates the cycle. When the state is initiator of iteration, then iteration is well-situated.*

Proof. The path from cycle to the appropriate final state leads through it, because traversal was evoked from the appropriate final state. □

One last detail has to be solved. I claim that RTE of state can be built in post-order, but I also claim that cycle is detected, when state is on the stack path, which means that it is a leaf of the recursion call tree. Thus I have to propagate this information from bottom to up, then I can distinguish which transition is self-loop and which is out-to-in. Let define set of back-links.

Definition 4.2.1 (Back-links)**.** Let back-links for current state u during recursion call tree is defined by following equations of dynamic programming on trees.

$$\text{Backlinks}(u) = \begin{cases} \{\text{Symbol}(u)\} & \text{For } u \text{ as current in rec. leaf call.} \\ \bigcup_{x \in \text{Kids}(u)} \text{Backlinks}(x) & \text{otherwise} \end{cases} \tag{4.1}$$

The main part of final algorithm is trivial. It is just calls subroutine for every final state and sum-ups the result.

The recursive subroutine firstly checks for termination condition (line 2), if condition is satisfied symbol of current state is added to back-links (line 3). Then it is collecting for RTEs from incoming transitions in two complementary sets (lines 5-20). Set of self-loop RTEs (loopEdges) and set of out-to-in RTEs (outToInEdges). Then RTE for current state is computed and returned (lines 21-26).

Algorithm 5 Dynamic programming approach

Input: NFTA $A = (Q, \sum, F, \delta)$.

Output: an equivalent RTE.

```
1: procedure ToRTE($A = (Q, \sum, F, \delta)$)
2:     $R \leftarrow \emptyset$
3:     for all $f \in F$ do
4:         stackPath ← new Stack()
5:         backlinks ← new Set()
6:         rte ← rteOfState($A$, $f$, stackPath, backlinks)
7:         $R \leftarrow R \bigcup \{$rte$\}$
8:     return $\sum_{x \in R} x$
```

Algorithm 6 Dynamic programming approach — rteOfState

Input: NFTA A[In], state s[In], stack path [In-Out], back-links [Out]

Output: an equivalent RTE.

```
1: procedure rteOfState($A = (Q, \sum, F, \delta)$, $s$, stackPath, backlinks)
2:     if $s \in$ stackPath then
3:         backlinks ← backlinks $\bigcup \{s\}$
4:         return $s$
5:     loopEdges ← $\emptyset$, outToInEdges ← $\emptyset$
6:     for all $t \in$ transitions to $s$ do
7:         edgeRTE ← new RTE()
8:         edgeRTE.setSymbol(Symbol($t$))
9:         doesLinkToCurrent ← false
10:        for all $c \in$ Kids($t$) do
11:            neighbourBacklinks ← new Set()
12:            kidRTE ← FinalToRTE($A$, $s$, stackPath $\bigcup \{$state$\}$,
                 neighbourBacklinks)
13:            edgeRTE.addKid(kidRTE)
14:            backlinks.insertAll(neighbourBacklinks)
15:            if $s \in$ neighbourBacklinks then
16:                doesLinkToCurrent ← true
17:        if doesLinkToCurrent then
18:            loopEdges.insert(edgeRTE)
19:        else
20:            outToInEdges.insert(edgeRTE)
21:    outToIn ← $\sum$ outToInEdges
22:    result ← outToIn
23:    if loopEdges.any() then
24:        loopRTE ← Iteration($\sum$ loopEdges, Symbol($s$))
25:        result ← Substitution(loopRTE, outToIn, Symbol($s$))
26:    return result
```

4.2.3 Time complexity

Let me analyse algorithm's time complexity for input automaton $A = (Q, \sum, F, \delta)$ Recursion terminates if current state is on the stack or when no other path exists. Assume automaton that is fully connected by transitions of unary symbols (from each state to each state). Then then all stack combination are reachable. Recursion starts from each final state. Thus final time complexity of algorithm is in $O(|F||Q|!)$.

Chapter 5

Implementation and Testing

Algorithm 4.2.2 is implemented as a part of Algorithms Library Toolkit (ALT).

5.1 Algorithms Library Toolkit

The ALT is an ultimate tool consisting of different kinds of automata, grammars, graphs, and other well-known structures in computer science. Moreover, a rich collection of algorithm related to these structures is implemented there.

ALT has been established by Ing. Jan Trávníček, Ph.D. at Faculty of Information Technology of Czech Technical University in Prague and powered by Ing. Tomáš Pecka, Ing. Štěpán Plachý. This project is continually maintained by collective of committees supervised by doctor Trávníček.

Its query language is suitable for integration testing as well as interactive simulations and debugging. An instance of structure can be created in place or loaded from a file. Then it can be stored in a variable or piped into an algorithm. The algorithm's result can be stored in a variable or stored in a file or piped into another algorithm. This brilliant interface allows creating testing pipeline as a chain of algorithms.

A unique advantages of this option is that RTE, as well as NFTA and DFTA, is already implemented with visualisation ability. Moreover, there are implemented helper algorithms to set up appropriate tests such as conversion in the opposite direction, determinization of NFTA, minimization of DFTA and equality comparison between two DFTAs.

5.2 FTA comparison

One kind of equality determination for two FTA is that languages accepted by automata are the same. Formally, let A and B be the FTAs, then they are equal if and only if $L(A) = L(B)$. This is mathematically correct, but technically comparison of two languages in a computer is impracticable since languages are usually infinite sets of trees. Nevertheless, every FTA can be converted into equivalent minimal form. The minimal form of FTA is variant cut-off states which are not leading to a final state, then determinized and minimized.

Furthermore, an FTA is similar to the graph structure. States are vertices and information whether state is final can be encoded in its name. Transitions are ”weird edges”. In a closer look, one transition consists of the incoming children to a ranked symbol. Then let me add one special vertex to the graph denoting the transition. Let incoming children be edges leading from a state into special the special vertex by simple transition with child’s index on it. Let ranked symbol be an edge leading from the special state to another state by transition with a symbol on it. Finally, an FTA can be considered as a directed graph with symbols on edges.

Moreover, comparison can be done be lemma 5.2.1, however, the definition of isomorphism has to be stated first.

Two graphs which contain the same number of graph vertices connected in the same way are called to be isomorphic.

Definition 5.2.1 (Graph isomorphism)**.** Two graphs G and H with graph vertices $V_n = 1, 2, ..., n$ are said to be isomorphic if there is a permutation p of V_n such that u, v is in the set of graph edges $E(G)$ if and only if $p(u), p(v)$ is in the set of graph edges $E(H)$.

Lemma 5.2.1. *If minimal forms of two FTA are isomorphic graphs, then they have to accept the same language.*

Proof. According to isomorphism (definition 5.2.1), graphs have same number of vertices and are connected in the same way. Thus automata are constructed in the same way. Obviously same-constructed automata have to accept same language. □

5.2.1 Random FTA generator

Another useful component for testing is a generator that provides random FTAs. The most significant advantage of input generator is that the generator requires just a tiny storage space than data it can generate. A right generator is based on the probability model, that allows generating all instances from the domain. Furthermore, a right-implemented generator provides the facility to set up an initial seed for the random generator. Generator initialised with the same seed should always behave in the same way. This is important for testing when a test failed on some specific input, the seed can be used to generate exactly the same input again for later analysis.

In ALT, there is already a generator for FTA, which seams to fulfils mentioned conditions for my purpose.

```
RandomTreeAutomatonFactory Ssiz Asiz MR Den
where
  Ssiz is number of states,
  Asiz is number of ranked symbols in alphabet,
  MR is maximal rank,
  Den is density between 0 to 1,
    1 is fully-connected automaton.
```

5.2.2 Testing pipeline

The testing pipeline can be constructed since all necessary parts have been introduced. Comparison of two FTA's is implemented in different approach as described in 5.2. Both automata are normalised firstly; this process reconstructs automata in a deterministic way, which makes comparison easier.

```
# Store minimized FTA
execute RandomTreeAutomatonFactory Ssiz Asiz MR Den >
   $inputFTA
execute $inputFTA
  | automaton::determinize::Determinize -
  | automaton::simplify::Trim -
  | automaton::simplify::Minimize -
  | automaton::simplify::Normalize - > $minimalOrigFTA

# Convert to RTE, back and compare
execute $inputFTA
  | automaton::convert::ToRTEDynamicProgramming -
  | rte::convert::ToFTAGlushkov -
  | automaton::determinize::Determinize -
  | automaton::simplify::Trim -
  | automaton::simplify::Minimize -
  | automaton::simplify::Normalize -
  | compare::AutomatonCompare - $minimalOrigFTA
```

5.3 Testing batches

This code describes testing pipeline for one sample automaton. In my testing routine I have three batches per 100 test samples.

5.3.1 1st batch - small and dense

The first batch contains small and dense automata. Density is set to 1, number of states is between 0 and 10, alphabet contains at most 5 ranked symbols and maximal rank is 4. This batch is the best for implementation and first debugging, because these automata are small enough to be solve by hand.

5.3.2 2st batch - medium-size and medium-density

The second batch consists of medium-size and medium-density automata. Density is set to 1, number of states is between 0 and 50, alphabet contains at most 10 ranked symbols and maximal rank is 4.

5.3.3 3st batch - large and sparse

The last batch is composed of large and sparse automata. Density is set to 0.2, number or states is between 0 and 120, alphabet contains at most 15 ranked symbols and maximal rank is 4.

5.4 Which solution is better?

In perspective of time complexity, states removal solution is surely better. Removal of state algorithm is polynomial, whereas dynamic programming solution is factorial in the worst case. Nevertheless, this is not the only criterion. Both algorithms give different results. These FTE are significantly different shaped. Let me focus on this in deeper look. I can't measure how far is a solution from optimal solution or how beauty tree is or how intuitive, nevertheless, I can evaluate how many edges and nodes have a tree to get rid off to become empty tree. I think the larger tree is usually less desirable than the smaller one as well as smaller tree is more understandable then larger one. Let me define the qualitative property of tree number of nodes.

Definition 5.4.1 (Number of nodes of a three)**.** Let t is a tree then $\#\text{nodes}(t)$ is number of nodes tree composes of.

This is not the only criterion I could use. Also, dept or some cumulative penalty for unbalance or a combination of more of these could be used. In the presented criterion, the tree node of any type has the same penalty. Also, this can be done in a different way, for example, bigger penalty for tree iterations.

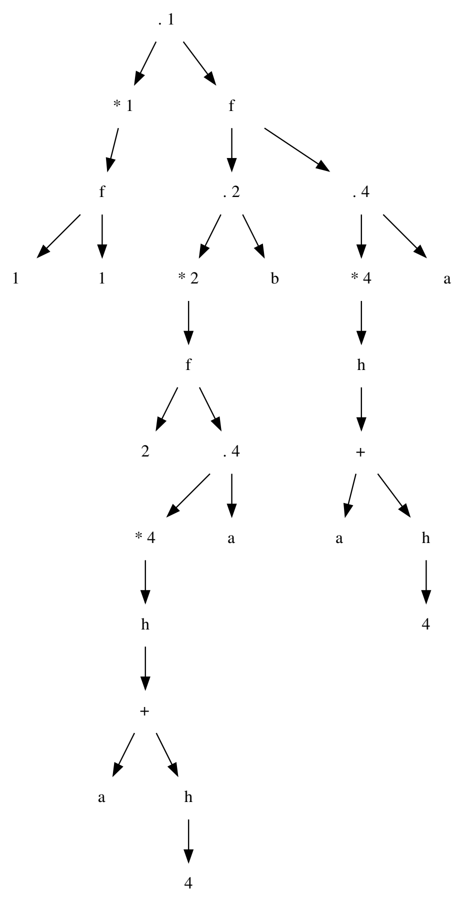


Figure 5.1: This RTE a is conversion result of automaton 2.1 processed by dynamic programming algorithm. #nodes(a) = 27

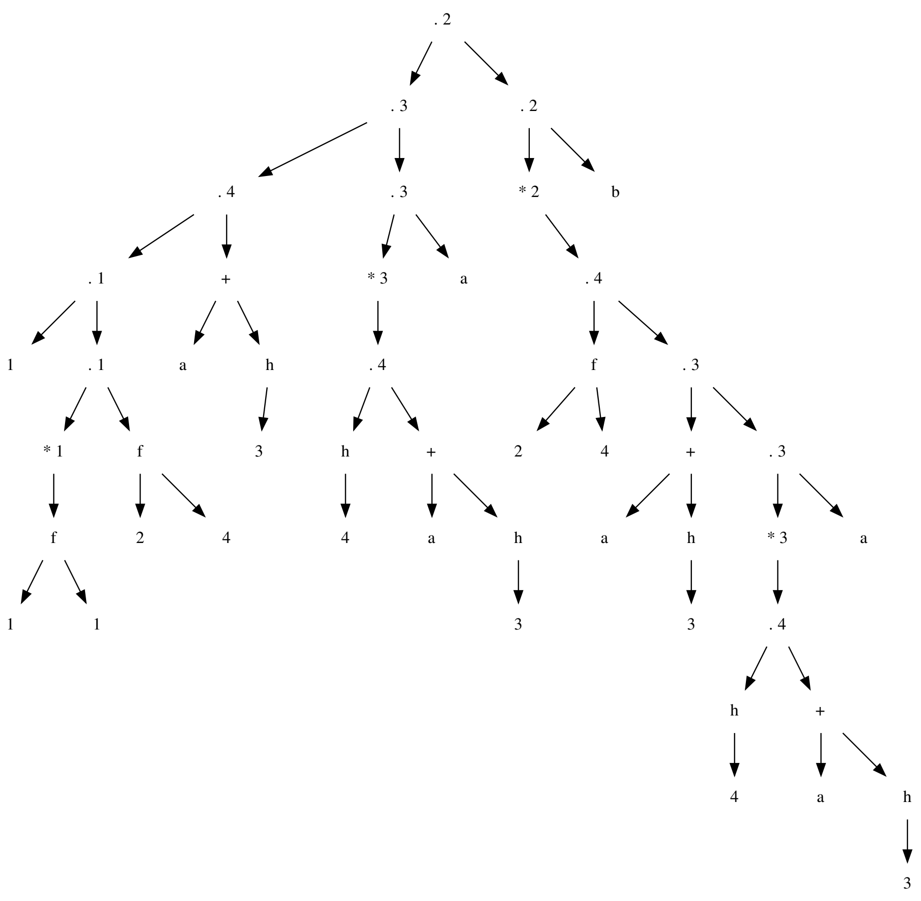


Figure 5.2: This RTE b is conversion result of automaton 2.1 processed by removal of state algorithm. $\#\text{nodes}(b) = 49$

The difference between the outputs of both algorithms is significant. Furthermore, note that used sample automaton is relatively small in size. Let me compare how qualitative property behaves over all three testing batches for both algorithms.

Following charts visualise results of measurements. In addition there is mean, standard deviation, 25th and 75th percentiles.

Measurement on 1st batch

For these inputs of this class, dynamic programming solutions is almost always better according to this criterion.

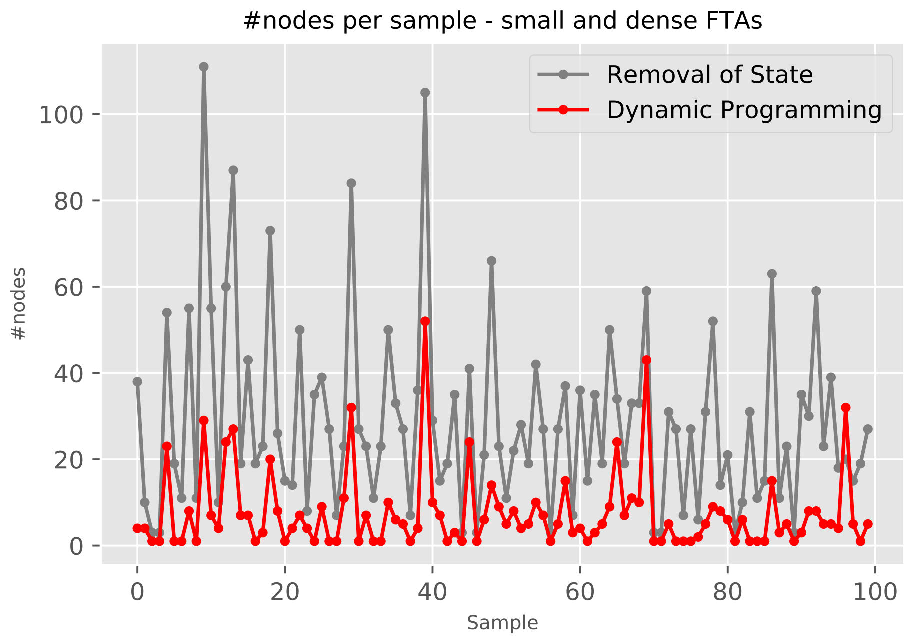


Figure 5.3: Comparison of algorithms on small dense automata.

mean	7.460000	28.620000
std	9.171123	21.562116
25%	1.000000	14.000000
75%	8.000000	36.000000

Measurement on 2nd batch

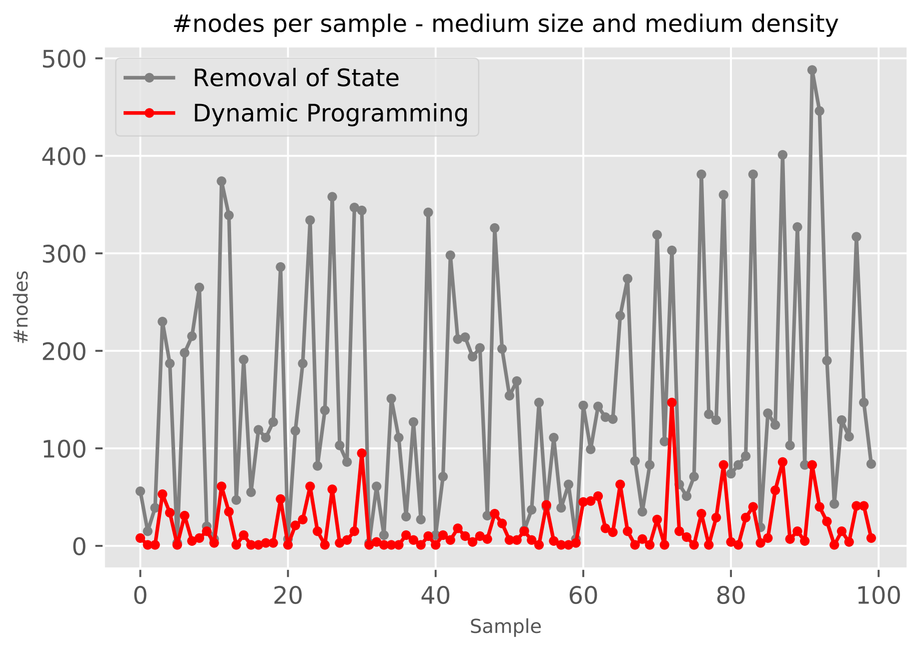


Figure 5.4: Comparison of algorithms on medium-size and medium-density automata.

For automata from the medium-size class, the dynamic programming solution seems to be more stable, whereas removal of state is alternating between peaks and troughs. This alternation is probably caused because of order sensitivity (mentioned in section 3.1.2).

mean	19.920000	154.580000
std	25.586328	119.291696
25%	3.000000	62.500000
75%	29.500000	214.250000

Measurement on 3rd batch

Here it forms the same effect as for 2nd batch, but even more notable. It seems that removal of state algorithm generates always a bigger solution; nevertheless, note that automata from this batch are sparser. Thus optimisation by substitution may be less relevant here.

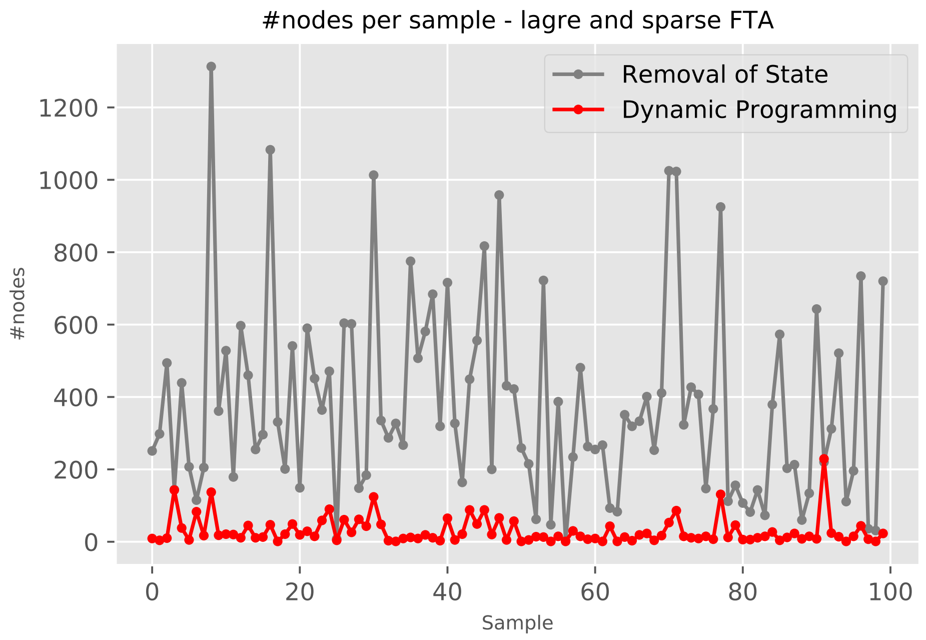


Figure 5.5: Comparison of algorithms on large and sparse automata.

mean	29.270000	378.460000
std	37.175493	267.743523
25%	7.750000	193.000000
75%	43.000000	510.500000

Conclusion

The analogy between textual languages and tree languages was mentioned. The problem of conversion regular tree automata to regular tree expressions was covered with a state-of-the-art solution by equation system. Furthermore, state elimination algorithm in the basic variant was designed and presented. Since there was no known algorithm to minimise an RTE, the quality of the result is considered as the secondary goal of this thesis. In spite of the algorithm is suitable for this work, two sources causing sub-optimal results were discovered. For each such an issue bespoke algorithm was proposed and implemented. Each algorithm was made to aim only one of the issues because no common solution resolving both issues simultaneously was found.

The testing pipeline was contracted for validation of correctness. Validation test was composed of 3 batches per 100 automata each. Relatively large (up to 120 states) automata are contained in batches as well as medium size (up to 50 states) and small (up to 10) automata.

One of the mentioned algorithms was the dynamic programming solution that was estimated as factorial in time complexity in the worst case. Another algorithm is called the removal of state that focuses on optimisation by substitution. This algorithm is polynomial in time. Dynamic programming solutions were able to compute all automata from testing batches, despite its estimated time complexity.

The qualitative measurable property of tree was defined for evaluation of outputted trees, that renders comparison between results possible. The comparison was showed and the dynamic programming approach was considered outstandingly better for bigger automata at least for mentioned testing batches. The remark was stated that it could be caused because of bigger automata were generated sparser and sparser automata could be optimised better by well-situated loops (dynamic programming approach) then by substitution.

Bibliography

[1] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications.* Available on: `http://www.grappa.univ-lille3.fr/tata`. release October, 12th 2007. 2007.

[2] V. Diekert and STACS 2004 M. Habib editors. *21st Annual Symposium on Theoretical Aspects of Computer Science.* release October, 12th 2007. Montpellier, France, 2004.

[3] F. Drewes. *Lecture notes on TREE AUTOMATA prepared by the participants of the course Formal Languages.* Available on: `https://www8.cs.umu.se/kurser/5DV023/VT09/final.pdf`. release October, 12th 2007. 2010.

[4] F. Drewes. *Lecture notes on TREE AUTOMATA prepared by the participants of the course Formal Languages.* Available on: `https://www8.cs.umu.se/kurser/5DV023/VT09/final.pdf`. release October, 12th 2007. 2010.

[5] J. Janoušek J. Trávníček and B. Melichar. *Indexing trees by pushdown automata for nonlinear tree pattern matching.* 2011.

[6] Bořivoj Melichar Jan Janoušek. "On regular tree languages and deterministic pushdownautomata". In: *Acta Informatica* (2009). Published online: 1 September ©2009 Springer.

[7] K. Knight and J. May. *Applications of Weighted Automata in Natural Language Processing Handbook of Weighted Automata.* 2009.

[8] W. Martens and J. Niehren. *On the minimization of XML Schemas and tree automata for unranked trees, Special Issue: Database Theory 2005.* 2007.

[9] T. Pecka. "Construction of a Pushdown Automaton Accepting a Language Given by Regular Tree Expression". MA thesis. Czech Technical University, Faculty of Information Technologies, 2016.

[10] Hadda Cherroun Younes Guellouma. “From Tree Automata to Rational Tree Expressions”. In: *International Journal of Foundations of Computer Science* 29.6 (2018).

Appendix A

Acronyms

ALT	Algorithms Library Toolkit
FA	Finite State Automaton
FTA	Finite Tree Automaton
DFTA	Deterministic Finite Tree Automaton
NFTA	Non-deterministic Finite Tree Automaton
RE	Regular Expression
RTE	Regular Tree Expression

Appendix **B**

Contents of enclosed CD

Attached disk contains whole ALT repository, the structure of project's file-system and naming are a bit tricky. Therefore, I am denoting position in project of main files of my work on the directory tree.

DP approach means algorithm based on Dynamic Programming approach.
RS approach means algorithm based on Removal of State approach.

```
readme.txt ...................... the file with CD contents description
└─ ALT........................the repository of Algorithms Library Toolkit
   └─ alib2algo
      └─ src
         └─ automaton
            └─ convert ............................... my implementation
               ├─ ToRTEDynamicProgramming.h ............. DP approach
               ├─ ToRTEDynamicProgramming.cpp ........... DP approach
               ├─ ToRTERemovalOfState.h ................. RS approach
               └─ ToRTERemovalOfState.cpp ............... RS approach
└─ text ......................................... the thesis text directory
   ├─ thesis.pdf.........................the thesis text in PDF format
   └─ src ................ the directory of LaTeX source codes of the thesis
```