FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Backreferences in practical regular expressions
Student:	Martin Hron
Supervisor:	Ing. Ondřej Guth, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2020/21

Instructions

Make a thorough research on existing algorithms for regular expression matching.
Focus on regular expressions with backreferences.
Upon agreement with the supervisor, implement a selected method based on memory automata defined in [1].
Compare this approach with existing regex matching tools.

References

[1] Schmid, M. Characterising REGEX languages by regular languages equipped with factor-referencing. In: Information and Computation. Volume 249, August 2016. ISSN 0890-5401. DOI 10.1016/j.ic.2016.02.003.

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

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

Prague January 5, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Backreferences in practical regular expressions

Bc. Martin Hron

Department of Theoretical Computer Science

Supervisor: Ing. Ondřej Guth, Ph.D.

May 27, 2020

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Guth Ph.D. for guiding this thesis and for providing all the valuable feedback and helpful advice. I would also like to thank my family for their support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 27, 2020 . . . . . .

Czech Technical University in Prague

Faculty of Information Technology

Citation of this thesis

Hron, Martin. *Backreferences in practical regular expressions.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Zpětné reference (odkazy) jsou rozšíření regulárních výrazů bežně podporované v dnešních nástrojích. Regulární výrazy se zpětnými referencemi mají zvýšenou vyjadřovací sílu, ale jejich vyhledávání (matching) je NP-úplné. Tato práce poskytuje přehled existujících přístupů pro vyhledávání regulárních výrazů se zpětnými referencemi a také teoretického výzkumu na toto téma. V rámci této práce byl implementován nástroj pro vyhledávání regulárních výrazů založený na modelu paměťových automatů (memory automata). Vyhledávání regulárních výrazů s počtem zpětných referencí na různé skupiny omezených konstantou pomocí paměťových automatů má polynomiání časovou složitost. Byla implementována další nedávno zveřejněná metoda založená na paměťových automatech, která poskytuje polynomiální složitost i pro výrazy s neomezeným počtem zpětných referencí splňujících jistou vlastnost. V rámci této práce byl navržen a implementován alternativní algoritmus pro výpočet této vlastnosti. Model paměťového automatu byl dále rozšířen pro podporu kvantifikátorů omezeného počtu opakování a další rozšíření byla implementována. Experimentální vyhodnocení ukázalo, že implementovaný nástroj je mnohem odolnější vůči katastrofickému backtrackování než existující implementace podporující zpětné reference. Žádný z testovaných útoků přes algoritmickou složitost nevyvolal znatelné zpomalení.

Klíčová slova Zpětné reference, Regulární výrazy, Regex, Paměťové automaty, Implementace regulárních výrazů

Abstract

Backreferences are an extension of regular expressions commonly supported in modern tools. Regular expressions with backreference have an increased expressive power but their matching problem is NP-complete. This work researches existing approaches for regular expression matching with focus on backreferences, and also provides an overview of theoretical work on the topic. As a part of this thesis, a matching tool based on computational model called memory automata was implemented. Matching patterns with number of backreferences to different groups limited by a constant using memory automata has polynomial time complexity. An additional recently published technique based on memory automata was also implemented, which provides polynomial complexity even for a subset of patterns with unbounded number of backreferences restricted by a certain property. As a part of this thesis, an alternative algorithm to compute this property was proposed and implemented. The memory automaton model was also extended to support counting constraints, and other common extensions were implemented. Experimental evaluation showed that the implemented tool is much more resistant to catastrophic backtracking when compared to existing implementations with support for backreferences. No tested algorithmic complexity attack triggered significant slowdown.

Keywords Backreferences, Regular expressions, Regex, Memory automata, Regular expressions implementation

Contents

List of Figures

List of Tables

Introduction

Regular expressions are a well established tool for description of regular languages in formal language theory. They are also of high practical relevance and their variants are used in wide array of applications, including text processing, search engines, data validation, databases, lexical analysis and simple parsing. For instance, many text editing applications allow their users to use regular expressions for tasks such as searching and replacing in text. Regular expressions are also used by various UNIX tools and utilities (for example grep, AWK and sed). Moreover, most modern programming languages support regular expressions, either as core functionality or via additional libraries.

While the theoretical concept has not changed much since its introduction, the practical regular expressions, as implemented in tools and libraries, have gained many extensions over time. Some of these extensions were merely added to simplify ease of use, serving only as a "syntactic sugar", and do not modify the expressive power and matching complexity. However, one extension – addition of the so called *backreferences*, which are now supported in most modern implementations – significantly increases the expressive power of practical regular expressions. Backreferences are a construct used to reference text matched by some previous part of pattern. They can be used to identify a repeated character or substring in the input text.

The most interesting problem for applications of regular expressions is the matching problem, i.e. to decide whether an input text belongs to the language described by given pattern. It can be shown, that the matching problem of regular expressions extended with backreferences is NP-complete. The added expressive power thus comes at the cost of significant increase of matching complexity of such patterns.

This work explores various techniques addressing this problem and reviews approaches to practical regular expressions matching in general. Some of the presented methods come with polynomial matching algorithms for certain subclasses of the problem.

An interesting model of computation for regular expressions with backreferences are memory automata, which were first introduced in [1]. Recent works on memory automata [2, 3] present number of polynomial matching techniques, for matching regular expressions with backreferences under certain restrictions. Therefore, it may be worthwile to implement a regex matching tool using memory automata and also employing selected improvements described in the recent articles.

Goal

The goal of this thesis is to research algorithms used for practical regular expressions matching, with strong focus on handling the backreferences, and then also implement regular expression matching tool based on memory automata. Then compare this approach with other existing regular expression implementations.

Organization of the thesis

First chapter provides the theoretical preliminaries needed for topics dealt with in this thesis. The second chapter introduces the concept of backreferences and provides an overview of different formalization of this regular expression extension. Overview of existing research into the theoretical properties of such patterns is also presented. The *regex* patterns, which are described by memory automata, are introduced in detail and their properties are described. Additionally, other common extensions found in practical regular expressions and the POSIX standard are discussed in the chapter.

The third chapter presents a research of existing approaches to regular expression matching, with focus on regular expressions with backreferences. A brief overview of existing implementations is provided at the end of the chapter.

The fourth chapter deals with the memory automata. The automaton model is introduced and its properties are outlined. Additionally, the recently published matching methods based on memory automata are discussed.

Chapter 5 describes implementation of the regular expression library based on memory automata that was created as part of this thesis. Modifications to the memory automaton model for practical regular expressions, especially for support of the counting constraints extension is presented. Automaton representation, construction, and the implemented matching algorithms are described.

Finally, Chapter 6 presents the results of experimental evaluation done on the implemented library. Comparison to other regular expression matching tools is provided.

Chapter 1

Preliminaries

This chapter provides an overview of mathematical preliminaries needed for topics covered in this work. The used notation is introduced and basic definitions of major concepts, such as the various computational models (automata) and their languages, are provided. The (classical) regular expressions, which are central to the subject of the thesis, are also defined in this chapter.

1.1 Basic notation

Let $\mathbb{N} = \{1, 2, 3, \ldots\}$ and $\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$. For $k \in \mathbb{N}, [k] = \{1, 2, \ldots, k\}$. For any set A, its *power set* is denoted by $\mathcal{P}(A)$, where $\mathcal{P}(A) = \{S \mid S \subseteq A\}$.

The standard notation for intervals is used, $[a, b] = \{x \mid a \leq x \leq b\}$, $(a, b) = \{x \mid a < x < b\}$, $[a, b) = \{x \mid a \leq x < b\}$, and $(a, b] = \{x \mid a < x \leq b\}$.

1.2 Alphabet, string and language

Definition 1.1 (alphabet)**.** An *alphabet* is a finite, nonempty set (commonly denoted by Σ). Elements of an alphabet are called *symbols*.

An alphabet of size 2 is called *binary* alphabet, and usually consists of {0,1}. An alphabet with only one symbol is called an *unary* alphabet. △

Definition 1.2 (string)**.** Finite sequence of symbols, where each symbol is an element of alphabet Σ, is called a *string* over alphabet Σ.

For a string w, the length of the string w, denoted $|w|$, is the number of symbols in w. Additionally, $|w|_a$ denotes the number of occurrences of symbol a in string w. Symbol ε denotes an empty string (i.e. string of length zero). For a string w and $i \in [|w|]$, $w[i]$ denotes its i-th symbol.

For a symbol $a \in \Sigma$ and $k \in \mathbb{N}$, a^k denotes string consisting of k repetitions of the symbol a.

For two strings $u = u_1 \ldots u_n$ and $w = w_1 \ldots w_m$, $u \cdot w$ (or simply uw) denotes their *concatenation*, that is the string $u_1 \ldots u_n w_1 \ldots w_m$.

Factor (or *substring*) of a string $w = w_1 \ldots w_n$ is any string $w_i \ldots w_j$, where $1 \leq i \leq j \leq n$. If $i = 1$ then it is also called a *prefix*, if $j = n$ it is called a *suffix*. △

Definition 1.3 (language)**.** A *language* L over an alphabet Σ is a set of strings over Σ.

For an alphabet Σ, the language containing all strings over Σ is denoted Σ^*. Language containing all non-empty strings is denoted Σ^+, therefore it holds that $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. For any alphabet Σ, we also set $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.

For any language descriptor[1] $\mathcal{D}$, the language described by $\mathcal{D}$ will be denoted using $\mathcal{L}(\mathcal{D})$. △

Definition 1.4 (operations on languages)**.** Let Σ_1 and Σ_2 be alphabets. For two languages L_1 over Σ_1 and L_2 over Σ_2, the following binary operations are defined:

- *union*, *intersection* and *difference* operations are defined using the respective standard set operation (for example the union is a language $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$ over alphabet $\Sigma_1 \cup \Sigma_2$),
- *concatenation* operation as $L_1 \cdot L_2 = \{u \cdot w \mid u \in L_1 \wedge w \in L_2\}$ over alphabet $\Sigma_1 \cup \Sigma_2$.

Additionally, for a language L over alphabet Σ, the following unary operations are defined:

- *n-th power* of a language recursively as $L^n = L^{n-1} \cdot L$, with $L^0 = \{\varepsilon\}$,
- *iteration* (or *Kleene star*) operation as $L^* = \bigcup_{n=0}^{\infty} L^n$ and *positive iteration* $L^+ = \bigcup_{n=1}^{\infty} L^n$,
- and *complement* of a language as $\overline{L} = \Sigma^* \setminus L$ over alphabet Σ.

△

[1] In the context of this thesis, language descriptors can be e.g. deterministic and nondeterministic finite automata, memory automata, regular expressions and their extensions.

1.3 Finite automata

Definition 1.5 (nondeterministic finite automaton)**.** A *nondeterministic finite automaton* (or *NFA* for short) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of *states*,
- Σ is an *input alphabet*,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is a *transition function*. Members of δ are called transitions, when $p \in \delta(q, a)$ (where $a \in \Sigma_\varepsilon$), such transition is called a-transition (specially ε-transition for $a = \varepsilon$). This transition can be denoted using $q \xrightarrow{a}_\delta p$, or just $q \xrightarrow{a} p$ when context is clear.
- $q_0 \in Q$ is a *start* state,
- $F \subseteq Q$ is a set of *final* (or *accepting*) states.

△

Definition 1.6 (configuration of NFA)**.** *Configuration* of a nondeterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ on input $w \in \Sigma^*$ are pairs (q, u), where $q \in Q$ and u is a (possibly empty) suffix of w. The configuration $(q_0,\ w)$ is called the *start* (or *initial*) *configuration* of M on w. A configuration (q, ε), where $q \in F$, is called an *accepting configuration*. △

Definition 1.7 (computation of NFA, language of NFA)**.** For a nondeterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$, input $w \in \Sigma^*$. The transition relation $\vdash_M$ is a relation on the configurations of M on w, such that $(q, u) \vdash_M (q', u')$ iff $u = au'$, where $a \in \Sigma_\varepsilon$ and $q' \in \delta(q, a)$.

An input $w \in \Sigma^*$ is *accepted* by M if $(q_0, w) \vdash_M^* (q, \varepsilon)$, where $q \in F$ (and $\vdash_M^*$ is the reflexive-transitive closure of $\vdash_M$). If an input is not accepted, we say it was *rejected* by M. A sequence of configurations $(q_0, w) \vdash_M (q_1, w_1) \vdash_M \ldots \vdash_M (q_n, w_n)$ is called a *computation* of M on w.

The language $\mathcal{L}(M)$ accepted by NFA M is defined as the set of all strings accepted by M, i.e. $\mathcal{L}(M) = \{w \mid w \in \Sigma^*, \exists q \in F, (q_0, w) \vdash_M^* (q, \varepsilon)\}$ △

By restricting the transition function to always return exactly one next state for any input symbol and disallowing ε-transitions, we gain the deterministic finite automaton.

Definition 1.8 (deterministic finite automaton)**.** A *deterministic finite automaton* (or *DFA* for short) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of *states*,

- Σ is an *input alphabet*,
- $\delta : Q \times \Sigma \to Q$ is a *transition function*
- $q_0 \in Q$ is a *start* state,
- $F \subseteq Q$ is a set of *final* (or *accepting*) states.

△

The semantics of DFA and its accepted language are defined similarly as for NFA. For any NFA, there exists a DFA accepting the same language, therefore the sets of languages that can be accepted by these two automaton models are equivalent [4].

1.4 Other automata models

Other, more expressive automata models are *push-down automata* and *Turing machines*, see [4] for their definitions.

A restricted form of Turing machine allowing to operate only on a tape of limited size (instead of infinite tape) is called a *linear bounded automaton* (*LBA* for short). It was first defined in [5], where only its deterministic version is introduced. The nondeterministic variant was established later in [6]. In this work, the term linear bounded automata (LBA) refers to the more general nondeterministic variant.

1.5 Language hierarchy

Definition 1.9 (language types)**.** We say that a formal language L is:

- *Recursively enumerable* (type 0), if there exists a Turing machine accepting it.
- *Context-sensitive* (type 1), if there exists a linear bounded automaton accepting it.
- *Context-free* (type 2), if there exists a pushdown automaton accepting it.
- *Regular* (type 3), if there exists a finite automaton accepting it.

△

1.6 Regular expressions

Regular expressions are another formalism for description of regular languages originally introduced by S. C. Kleene [7]. To differentiate them from the extensions that will be introduced later, they are referred to as *classical regular expressions* throughout the text. Conversely, the extensions, i.e. the variants of regular expressions implemented in real-world tools and libraries, will generally be referred to as *practical regular expressions*, the various dialects and formalisms will be discussed later.

Definition 1.10 (classical regular expression)**.** Let Σ be an alphabet, then the set RE_Σ of *(classical) regular expressions over* Σ, and for each $\alpha \in \mathrm{RE}_\Sigma$ the described regular language $\mathcal{L}(\alpha)$, are defined recursively by the following rules:

1. For every $a \in \Sigma_\varepsilon$, $a \in \mathrm{RE}_\Sigma$ and $\mathcal{L}(a) = \{a\}$.
2. $\emptyset \in \mathrm{RE}_\Sigma$ and $\mathcal{L}(\emptyset) = \emptyset$.
3. For every $\alpha, \beta \in \mathrm{RE}_\Sigma$:
   a) $(\alpha \cdot \beta) \in \mathrm{RE}_\Sigma$ and $\mathcal{L}((\alpha \cdot \beta)) = \mathcal{L}(\alpha) \cdot \mathcal{L}(\beta)$,
   b) $(\alpha \vee \beta) \in \mathrm{RE}_\Sigma$ and $\mathcal{L}((\alpha \vee \beta)) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$,
   c) $(\alpha)^+ \in \mathrm{RE}_\Sigma$ and $\mathcal{L}((\alpha)^+) = \mathcal{L}(\alpha)^+$,
   d) $(\alpha)^* \in \mathrm{RE}_\Sigma$ and $\mathcal{L}((\alpha)^*) = \mathcal{L}(\alpha)^*$.

Note, that the definition uses operations on languages from Definition 1.4. △

The set of languages that can be described by classical regular expressions is exactly the set of regular languages [4].

1.7 Other preliminaries

Various other concepts from algorithmic complexity and formal language theory will be used. Overviews are provided in [8] and [4] respectively.

Chapter 2

Regular expressions with backreferences

This chapter introduces the concept of backreferences in regular expressions. It is important to note that the terminology is not very standardized. Different terms such as regex (or REGEX), extended regex, regexp, rewbr and so on are used for practical regular expressions and their extensions, as well as for the various formalisms and derived language classes.

From now on, the term regular expressions with backreferences will be sometimes abbreviated as *rewbr*. Note that this usage is more general than e.g. in [9], where it denotes particular concrete variant of regular expressions. Here, the term will be used to refer to the general concept, the concrete definitions and their comparison are presented later in this chapter. Overview of real-world implementations is provided at the end of Chapter 3.

The concept is first introduced informally and illustrated on examples. Overview of works formalizing rewbr and performing theoretical analysis of their language classes follows. The definition of regex patterns used for memory automata is introduced in detail. Theoretical results about the known properties of the regex language class and its matching problem are also explored. Then, other extensions (than backreferences) found in practical implementations are overviewed briefly. The POSIX standard and its definition of regular expressions are also described at the end of the chapter.

2.1 Introduction

The backreference construct in regular expressions allows to match part of input text previously matched by some subpattern. In most implementations, pairs of parentheses "(" and ")" are used to mark the subpattern, whose matched text will be saved and can be referenced (matched again) later. The

term *capturing group* is also used as a synonym for the parenthesized subpattern and capturing group *contents* refers to the part of input text matched by this subpattern.

The backslash character followed by a number (for instance "\1") is then used to reference the text most recently matched by subpattern in the corresponding pair of parentheses. Parentheses pairs (and the corresponding capture groups) are usually given numbers according to the order of the opening (left) parenthesis in the pattern.

Example 2.1. Pattern[2] "`(a*)b\1`" matches strings from context-free language $L_1 = \{a^n b a^n \mid n \in \mathbb{N}_0\}$. The "`\1`" in this pattern is a reference to contents of the first *capture group*, that is the part of input most recently matched by subpattern inside the first pair of parentheses. The subpattern "`a*`" matches the symbol `a` any number of times, then the backreference "`\1`" matches the same string again. For instance, the input "aabaa" gets matched by this pattern, because the first subpattern matches "aa", then "b" is matched and then the backreference matches "aa" again. ⧫

Example 2.2. Another example of a more complicated pattern with multiple backreferences is: "`(a+(b*)a)\2(c+)\1\3`". Because parentheses pairs (capture groups) are numbered based on the order of left opening parenthesis, the backreference "`\2`" refers to text matched by subpattern "`b*`". The language described by this pattern is $L_2 = \{a^m b^n a b^n c^k a^m b^n a c^k \mid m, k \in \mathbb{N}, n \in \mathbb{N}_0\}$. ⧫

There are many variants (*dialects*) of rewbr in different environments and programming languages, see [10] and [11] for a good overview of practical regular expressions and their dialects. The referenced books also contain many examples of practical usage of rewbr patterns.

2.2 Formalizing backreferences

Several works attempt to formalize practical regular expressions and explore properties of the resulting language class. This section provides an overview of rewbr definitions and relationships between them, see also [12] for another overview. Same as there are differences between how various implementations match, definitions that were introduced also vary in semantics.

2.2.1 Early formalization using variables

An early attempt to formalize regular expressions with backreferences was in [9], however the semantics are described only informally. Here, a concept

[2] For these examples, syntax described in previous paragraphs is used. Such patterns are valid in common implementations – e.g. Java, PCRE, Python.

of *variables* is used. The capturing of a subpattern corresponds to variable assignment of the form "$(\alpha)\%v_i$", where α is rewbr (subpattern) and v_i is a variable from the set of variable names. The contents of a variable v_i is then referenced using "v_i" in the pattern. For instance, the Example 2.1 rewritten using this syntax would be: "`(a*)`$\%v_1$`b`$\,v_1$", where the text matched by "`(a*)`" is first assigned to variable v_1 and then recalled later.

Note that the definition in [9] is semantically different from how rewbr were (informally) described in Section 2.1, where the captured string was referenced by number of the corresponding parenthesis and there is always at most one subpattern contained in the k-th parentheses pair. In contrast, the definition with variables allows any number of variable redefinitions for different subpatterns. For example, in the pattern "`(a*)`$\%v_1$`c`$\,v_1$`(b+)`$\%v_1\,v_1$", variable v_1 is first assigned the string matched by "`(a*)`" subpattern, then recalled and then later reassigned to the string matched by "`(b+)`" subpattern and recalled again with the new value. This pattern thus describes language $\{a^n c a^n b^{2m} \mid n \in \mathbb{N}_0, m \in \mathbb{N}\}$. Differences between these two approaches are explored further below after introduction of precise definitions. Note also, that some implementations support named capturing groups that resemble variables discussed in this section. However, redefinition of group contents is usually not allowed, i.e. one name can be associated with at most one capturing group.

2.2.2 Formalization of rewbr with numbered backreferences

A precise definition of rewbr using parse trees was introduced in [13]. Unlike the definitions in [9] and [1] (explored later), this definition does not use variables, but instead it uses numbered backreferences to pairs of parentheses, much like examples from Section 2.1. As [3] points out, this approach was likely motivated by the implementations (Perl) at the time, where only numbered backreferences were supported.

Another noteworthy property of the definition in [13] is that it explicitly requires the referenced subpattern to be closed – i.e. the closing parenthesis belonging to the k-th pair of parentheses must precede any backreferences of the form "$\backslash k$". For instance, the pattern "`(a*\1)b`" is invalid.

Note, that because recursively defining rewbr with the above constraint seemed impossible in a convenient way [13], another auxiliary definition of patterns called *semi-regex* was also introduced. The set of patterns with the described property was then called *extended regex*.

The article [13] also provides pumping lemma for languages that can be described by such extended regex patterns. The authors also prove that the class of these languages is a proper subset of contex-sensitive languages and is incomparable with context-free languages. Another article [14] explores

properties of such languages further and also comes with an improved pumping lemma that restricts the class of languages recognized by extended regex even more.

2.2.3 Formalizing rewbr via factor-referencing

Another definition using factor-referencing was introduced in [1] and later used and expanded upon in works dealing with memory automata such as [3] and [2]. The rewbr patterns defined in these works are called *regex*, from now on, the term *regex* will be used to refer to this variant of rewbr patterns and the term *regex languages* will denote the class of languages that can be described by regex patterns. Since regex are central to memory automata, an exact definition of their syntax is provided below. See also [3] for the motivation behind this new definition and discussion of differences from the definition in [13].

The following definition was adapted from the appendix of [2], but [1] and [3] define the concept similarly.

Definition 2.1 (regex syntax)**.** Let Σ be an alphabet and let X be a finite set of *variables*. The set $\mathrm{RX}_{\Sigma,X}$ of *regular expressions with backreferences over* Σ *and* X, or *regex* for short, is defined recursively as follows:

1. For every $a \in \Sigma_\varepsilon$, $a \in \mathrm{RX}_{\Sigma,X}$ and $var(a) = \emptyset$.
2. For every $\alpha, \beta \in \mathrm{RX}_{\Sigma,X}$:
   a) $(\alpha \cdot \beta) \in \mathrm{RX}_{\Sigma,X}$ and $var((\alpha \cdot \beta)) = var(\alpha) \cup var(\beta)$,
   b) $(\alpha \vee \beta) \in \mathrm{RX}_{\Sigma,X}$ and $var((\alpha \vee \beta)) = var(\alpha) \cup var(\beta)$,
   c) $(\alpha)^+ \in \mathrm{RX}_{\Sigma,X}$ and $var((\alpha)^+) = var(\alpha)$.
3. For every $x \in X$, $x \in \mathrm{RX}_{\Sigma,X}$ and $var(x) = \{x\}$.
4. For every $\alpha \in \mathrm{RX}_{\Sigma,X}$ and $x \in X \setminus var(\alpha)$, it holds that $x\{\alpha\} \in \mathrm{RX}_{\Sigma,X}$ and $var(x\{\alpha\}) = var(\alpha) \cup \{x\}$.

Because it is desirable that regex be able to describe an empty language (so that it is a proper extension of classical regular expressions), $\emptyset$ will also be allowed as a regex, but occurrence of $\emptyset$ as a subpattern in other regex is not allowed. This rule is present in [3], but it is missing from the definition in [2]. This is likely a mistake in the latter article.

Additionally, α^* is a shorthand for $(\alpha^+ \vee \varepsilon)$, where $\alpha \in \mathrm{RX}_{\Sigma,X}$. Furthermore, the parentheses may be omitted, in that case '$+$' and '$*$' takes priority over '$\cdot$', which in turn takes priority over '$\vee$'. The '$\cdot$' may also be omitted, the absence of explicit operator between elements is called *juxtaposition*. For example $(\mathtt{a} \vee ((\mathtt{b}^+) \cdot \mathtt{c}))$ may be written as $\mathtt{a} \vee \mathtt{b}^+\mathtt{c}$. △

Using only[3] rules 1 and 2 yields a subset of $\mathrm{RX}_{\Sigma,X}$ that is equal to the classical regular expressions defined in Section 1.6. For $x \in X$, an expression of the form $x\{\alpha\}$ (see rule 4) is called a *binding of variable* x. An expression x (from rule 3) is then called a *reference to variable* x (or also a *recall of variable* x). The term *occurrence of variable* can denote both binding and reference of variable.

Example 2.3. Some examples of valid regex patterns over $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and $X = \{x_1, x_2\}$ are: $\alpha = x_1\{\mathtt{a}^+\}\mathtt{b}x_1$, or $\beta = x_1\{(x_2\{\mathtt{a}^+\}\mathtt{b}x_2 \vee \mathtt{b}^+)\}\mathtt{c}x_1^+$. Additionally, according to the definition of syntax, references to variables may precede bindings of variable (if any), so a pattern $\gamma = (x_1\mathtt{a}x_1\{\mathtt{b}^+\})^+$ is also valid.

On the other hand, because rule 4 (variable binding) does not allow the variable to also appear inside the bounded subexpression, patterns such as $\eta = x_1\{x_1\{\mathtt{a}^+\}\}$, or $\nu = x_1\{\mathtt{a}^+x_1\}$ are *not* valid regex. ♦

Now the regex semantics are concisely established here, see [1] for a thorough definition. The definition uses concepts of ref-word and ref-languages, which will be introduced briefly.

Definition 2.2 (regex semantics)**.** For a finite set of variables X, we define a set containing pair of opening and closing brackets for each variable as $\Gamma = \{\vdash_x, \dashv_x \mid x \in X\}$.

Then, for any $\alpha \in \mathrm{RX}_{\Sigma,X}$, the *ref-version of* α, denoted α_{ref}, is a classical regular expression defined as the result of recursively replacing all variable bindings $x\{\beta\}$ in α by $\vdash_x \beta \dashv_x$. The *ref-language of* α is then defined as[4] $\mathfrak{R}(\alpha) = \mathcal{L}(\alpha_{ref})$. A string $w \in \mathfrak{R}(\alpha)$ is a *ref-word*.

For a ref-word $w \in \mathfrak{R}(\alpha)$, the *dereference* $\mathcal{D}(w)$ is defined as w with every occurrence of any variable $x \in X$ replaced by string β, where $\vdash_x \beta \dashv_x$ is the last occurrence of matching $\vdash_x, \dashv_x$ pair preceding the x that is being replaced. The brackets are then also omitted from dereference. If there is no preceding $\vdash_x, \dashv_x$ pair, x is replaced by ε (ε-semantics, see below).

And finally, for a regex $\alpha \in \mathrm{RX}_{\Sigma,X}$, the language described by α is defined to be $\mathcal{L}(\alpha_{ref}) = \{\mathcal{D}(w) \mid w \in \mathfrak{R}(\alpha)\}$. △

Thanks to the way regex are defined, all $w \in \mathfrak{R}(\alpha)$ are well-formed with respect to each pair of brackets ($\vdash_x, \dashv_x$ for every $x \in X$), and no x occurs inside the corresponding bracket pair. The above definitions are demonstrated on following example with pattern corresponding to the one in Example 2.1.

[3] Along with the rule for $\emptyset$ to allow describing an empty language.

[4] $\mathcal{L}(\alpha_{ref})$ is the language described by classical regular expression α_{ref} over the alphabet $(\Sigma \cup X \cup \Gamma)$.

Example 2.4. For a regex pattern $\alpha = x\{\mathtt{a}^*\}\mathtt{b}x$ (where $\alpha \in \mathrm{RX}_{\{\mathtt{a},\mathtt{b}\},\{x\}}$), its ref-variant is $\alpha_{ref} = {\Rsh}_x\ \mathtt{a}^*\ {\Lsh}_x\ \mathtt{b}\,x$, where α_{ref} is a classical regular expression over alphabet $\{\mathtt{a}, \mathtt{b}, x\} \cup \Gamma$ and $\Gamma = \{{\Rsh}_x, {\Lsh}_x\}$.

Then $\mathfrak{R}(\alpha) = \mathcal{L}(\alpha_{ref}) = \{{\Rsh}_x\ \mathtt{a}^k\ {\Lsh}_x\ \mathtt{b}\,x \mid k \in \mathbb{N}_0\}$ is the ref-language of α. For instance $w = {\Rsh}_x\ \mathtt{aaa}\ {\Lsh}_x\ \mathtt{b}\,x$ is a ref-word from $\mathfrak{R}(\alpha)$ and by replacing x with $\mathtt{aaa}$ from inside the preceding brackets pair, we gain its dereference: $\mathcal{D}(w) = \mathtt{aaabaaa}$.

Finally, the language described by regex α is $\mathcal{L}(\alpha) = \{a^n b a^n \mid n \in \mathbb{N}_0\}$, which is equal to the language L_1 from Example 2.1. ♦

As seen from the example, this formalism is similar to the definition using variables from Section 2.2.1. Much like the definition in [9], this definition also allows multiple variable redefinitions using different patterns. Another issue discussed in [3] is how to treat references to undefined variables. For instance, in regex $\alpha = \mathtt{a}x\mathtt{b}^+x\{\mathtt{a}^+\}$, the variable x is referenced before it is assigned a value. Two natural options for default values of such undefined variables are ε and $\emptyset$, called *ε-semantics* and *$\emptyset$-semantics* respectively.

Under $\emptyset$-semantics, subexpressions with undefined variables will not match, i.e. they represent an empty language. In some implementations, undefined variable will cause an error, this behavior can be considered $\emptyset$-semantics as well. The *ε-semantics* are used in both [13] and [1], as well as in this thesis. Conversely, the *$\emptyset$-semantics* are used for instance in [9] and [14].

See [12] for detailed comparison of semantics between different definitions (and also some practical implementations) of rewbr, and for some additional pumping lemmas for various rewbr language classes.

2.3 Properties of the regex language class

This section deals with properties of languages that can be described by regex expressions defined in Section 2.2.3. The class of formal languages that can be described by a regex pattern will be called *regex languages*.

2.3.1 Relation to other language classes

Trivially, regular languages are a proper subset of regex languages, since any regular language can be described by a classical regular expression and every classical regular expression is also a regex. Additionally, regex are clearly more semantically powerful, Example 2.4 contains a language that is not regular but there is a regex pattern describing it.

Properties of languages described by a variant of rewbr patterns are explored in [13]. The main results are that such languages are a proper subset

of context sensitive languages and they are incomparable with the family of context-free languages. However, their definition (discussed in Section 2.2.2) of rewbr is semantically different from regex. As noted in [3], there are languages that cannot be described by such rewbr patterns but can be described by regex, i.e. the class of languages from [13] is a proper subset of regex languages. This fact is also proven in [12].

However, the above properties hold for regex languages too. There is no regex pattern for the context-free language $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$, because when matching the a^n, only substrings of a^n can be possibly saved in capture group and then recalled, but we need to match b^n, i.e. we would need to store the number of repetitions (instead of substrings) and use it to control repetitions of another subpattern, which can not be done using backreferences. Conversely, the language $L_2 = \{ww \mid w \in \{a, b\}^+\}$ can be described by regex pattern $\alpha = x\{(\mathtt{a} \vee \mathtt{b})^+\}x$, and L_2 does not belong into the family of context-free languages. Therefore, regex languages are incomparable with context-free languages.

The property of regex languages being a proper[5] subset of context-sensitive languages stems from the fact that any regex language can be accepted by a linear bounded automaton. This can also be shown by proving that the matching problem can be solved by nondeterministic Turing machine in linear space. This may be done by backtracking as described in [9], and since language class of their rewbr patterns is equivalent to regex languages [12], this can be used for regex as well. Another proof using memory automata is provided in Section 4.3.

2.3.2 Closure under operations

It is proved in [1] that the regex language class is closed under intersection with regular languages. The same article also claims that: *"It is known that REGEX languages are closed under union, but not under intersection or complementation"* and they cite [13] and [14] for proofs. However, as was already discussed, the formalism used in [13] has weaker semantic power than regex. Therefore, although [13] proves that their rewbr language class is not closed under complementation, and [14] proves it is not closed under intersection, an extended pumping lemma for regex class would be needed to explicitly prove that those properties also hold for regex languages. The pumping lemmas provided in [13], [14] (and also [12]) are not directly usable for the regex language class. It is admitted in [3] by one of the authors that using [13] as a reference in [1] and [15] is not entirely correct, because the restrictions from [13] are not used there.

[5] There is a context-free (and therefore also context-sensitive) language $\{a^n b^n \mid n \in \mathbb{N}\}$ that can not be described by regex, so regex languages can not be equivalent to the class of context-sensitive languages.

The closure under union property is trivial though, given any two regex languages L_1 and L_2, we can by definition construct regex patterns α and β such that $\mathcal{L}(\alpha) = L_1$ and $\mathcal{L}(\beta) = L_2$. The union $L_1 \cup L_2$ can then be described by regex pattern $(\alpha \vee \beta)$ and is therefore also a regex language.

2.4 The regex matching problem

Let $\alpha \in \mathrm{RX}_{\Sigma,X}$ be a regex pattern. The regex matching problem is to decide, for a given string $w \in \Sigma^*$, whether $w \in \mathcal{L}(\alpha)$.

The early analysis of the complexity of this problem was in [9], where a reduction from vertex-cover problem was used to prove that the problem is NP-complete. This proof uses large alphabet, however, as proved in [12] and [14], the problem is NP-complete even for unary alphabet and this holds for all rewbr variants described in Section 2.2.

Another interesting proofs[6] using reductions from 3-CNF-SAT and graph colorability on Perl regular expressions, can be found at [16]. As [3] notes, the NP-completeness of the problem also follows from complexity of analogous problem for Angluin patterns [17].

Complexity of the problem under various restrictions, such as on number of variables, number of occurrences of one variable, or alphabet size, are explored in [18]. Under many of these restrictions, the problem is still NP-complete. However, when the number of backreferences to different capturing groups is bounded by a constant, the resulting matching problem is in P [15]. This will be explored in detail later, in the next chapter.

Complexities of other (than matching) problems for the regex are investigated in [19]. The paper also shows that it is not possible[7] to effectively minimize regex patterns in regards to pattern length nor number of variables. Specially, deciding whether for a given k-variable regex there is an equivalent $(k-1)$-variable regex is undecidable [19].

2.5 Submatch addressing and searching in text

The matching problem described in previous section is a decision problem, its solution is a logical value: *True* or *False*. In practice, it may be desirable to have more information on the match, for instance which part of input got matched by a particular subpattern, this is called *submatch addressing*. For instance, the pattern "`.*(<.*>).*`" (using '`.`' as wildcard) would match any substring enclosed between '`<`' and '`>`' in input text. The solution to

[6] The authors note that only NP-hardness is shown there. For a problem to be NP-complete, it must also be shown that it is in NP, which is trivial, see [9].

[7] That is unless P=PSPACE.

matching problem as described in previous section would just be *True* if any such substring was found in the input, but using submatch addressing would also provide the substring that was matched by the "`<.*>`" subpattern.

Related problem is *searching* for regular expression in given input. That is given a regular expression pattern α, find a substring s (or possibly all such substrings) in input, such that $s \in \mathcal{L}(\alpha)$.

Associated issue is that there can be multiple possible matches in given input. Specific regular expressions implementations then must decide which of these matches will be reported. For instance, POSIX regular expressions use the leftmost longest rule [20].

2.6 Other extensions in practical regular expressions

This section provides a brief overview of some other extensions (than backreferences) typically found in practical regular expressions.

Most practical implementations support wildcard character (usually '`.`') to match any symbol, as well as many other constructs to express sets of possible symbols (like character classes, character sets, ranges etc.). For instance the pattern "`[ab[:digit:]]`" is a bracket-expression in POSIX syntax (see next section) matching either `a`, `b` or any symbol from the *digit* character class (i.e. any digit). An equivalent expression using character range instead would be "`[ab0-9]`".

Additionally, engines usually support variety of *assertions* in patterns. Assertion is a special symbol in pattern that does not consume any input characters, but it ensures some property in the input string. For instance, very common type of assertions are start of string and end of string assertions, usually denoted with '`^`' and '`$`' respectively. These can be used when searching in text to enforce that the match occurs in the beginning or at the end of given input. For example, the pattern "`a*b$`" in PCRE matches substrings of the form $\texttt{a}^n\texttt{b}$, but only at the end of input[8]. These assertions are also called *anchors*. Other assertions can be found in practice, such as word boundary assertion, which matches at either a beginning or an end of a word in string (as defined by the particular implementation). Even more complicated assertions such as look-ahead and look-behind, which allow to call another pattern as part of their constraint, are supported in some engines.

Another common feature are *counting constraints* (also called *bounded repetitions* or *counters*), which allows to repeat a subpattern number of times given by constant or in a form of range. For example, the pattern "`a{1,3}`" (in

[8] In multiline mode '`$`' also matches any (internal) end of line, analogously for '`^`'.

POSIX ERE syntax) matches input `a`, `aa`, or `aaa`. This feature does not extend semantic power of regular expressions, equivalent patterns can be constructed using alternation.

2.6.1 Subpattern recursion

An extension that further significantly increases the expressive power of regular expression patterns is *subpattern recursion*. It allows to call (reference) another subpattern from within the same pattern. Recursion is not as widely supported feature as backreferences, but it is for instance supported in PCRE library. Using PCRE syntax, the pattern "`(a(?1)?b)`" describes language $\{a^n b^n \mid n \in \mathbb{N}\}$. The "`(?1)`" construct[9] is a recursion (reference) to the first subpattern.

To the knowledge of author, regular expressions with recursion operator and their properties are not very well researched from the theoretical perspective. Power of regular expressions with both backreferences and subpattern recursion is investigated in [21], where it is claimed that such patterns are able to express all context-free languages. An intuitive proof is provided by showing how to construct recursive pattern for any production rule of context-free grammar [21]. Additionally, the exact relation between languages of such patterns and context-sensitive languages is unclear.

2.7 The POSIX standard

POSIX (Portable Operating System Interface) [20] is a family of standards that define a standard operating system interface and environment, along with command line shells and utility programs. Regular expressions are part of the POSIX standard and they are supported in major utilities, such as in grep, sed and awk command line tools.

Two different variants of regular expression syntax are part of the POSIX standard: the *basic regular expressions (BRE)* and *extended regular expressions (ERE)* syntax. The BRE variant is used in grep and sed utilities, as well as in some other tools like emacs editor. The ERE syntax is used for instance in awk and egrep utilities, some other utilities also support this syntax when called with additional flags. See [20] for detailed definitions of syntax and semantics for both variants.

BRE supports basic constructs like concatenation (using juxtaposition), Kleene star (using '`*`' operator), and element repetitions, using either '$\{m\}$' to repeat preceding element (subpattern) exactly m times, or '$\{m, n\}$' for range. Special character '`.`' acts as a *wildcard* to match any character (possibly

[9] The '`?`' after "`(?1)`" in "`(?1)?`" indicates an optional subpattern and is unrelated to recursion syntax.

excluding some specific characters like newline, depending on the application). Characters '`^`' and '`$`' are used to match starting and ending position in input respectively (as described in Section 2.6).

Additionally, bracket-expressions are supported, which can be used to match character sets, character ranges, character classes, etc. BRE syntax does *not* support alternation between subpatterns[10], although some applications (for example emacs) extend the syntax to support it, possibly alongside other extensions. Parentheses '`(`' and '`)`' can be used to mark subpatterns. BRE syntax also supports backreferences, expression '$\backslash k$' matches substring previously matched by the k-th subpattern.

The ERE syntax supports all constructs BRE (with the exception of backreferences, see below) and adds additional features. ERE additionally supports alternation (using '`|`' operator), positive iteration (using '`+`') and '`?`' for matching the preceding element one or zero times.

Another major difference is that the ERE variant defines more characters as special characters, so that they do not need to be escaped. For instance, the BRE pattern "`\(a*\)\{5\}b`" can be written simply as "`(a*){5}b`" in ERE syntax.

Unlike BRE, the ERE syntax does not contain backreferences, but compatible extensions may add support for them. Other extensions such as look-ahead, look-behind and pattern recursion are not part of POSIX regular expressions.

[10] Alternation between single characters can be achieved using the bracket-expression.

Chapter 3

Regular expression matching algorithms and approaches

This chapter provides an overview of approaches and methods for implementing practical regular expressions, focusing on backreference support in particular. Various techniques of how to deal with the complexity caused by adding backreferences (see Section 2.4) are explored. Implementations may be called regular expression *engines*. The main focus will be on the matching problem, i.e. to decide whether input text belongs to language described by pattern. Brief overview and comparison of some real-world regular expression engines is also provided at the end of the chapter.

See also [22] for a good overview of various techniques for implementing regular expressions. However, it largely focuses on non-backtracking implementations, without support for backreferences.

3.1 Thompson's algorithm

One of the oldest implementations of regular expressions is due to Ken Thompson, it is described in [23], published in 1968. Here, Thompson converts a regular expression into machine code instructions[11], in a somewhat similar fashion as virtual machine or JIT compilation approaches discussed later.

Naturally, this implementation did not support backreferences, as they are a concept introduced several decades later. The implementation supports classical regular expressions, which were established by Kleene in 1956 [7].

While it was not formalized there, the article provides a basis for transformation of regular expressions into finite automata. The conversion algorithm, known as Thompson's construction algorithm, is described in detail for in-

[11] IBM 7094 machine code to be specific.

stance in [4]. Given a classical regular expression, the algorithm constructs a NFA (nondeterministic finite automaton) accepting the same language.

Thompson's algorithm is one of several methods for regular expression to NFA transformation. Other examples of such algorithms are Glushkov's incremental construction algorithm [24] and Brzozowski's method of derivatives [25].

3.2 Basic approaches

Since there is an algorithm for constructing NFA from regular expression, it seems straightforward to decide the matching problem by first constructing the automaton and then simulating its computation for given input.

Depending on which type of automaton model is used and how it is simulated, there are three basic approaches to regular expression matching, they are described in the following subsections.

3.2.1 DFA based

For any given (classical) regular expression, a NFA can be constructed, using e.g. Thompson's construction. The resulting nondeterministic finite automaton can then be transformed into an equivalent[12] deterministic finite automaton using the *subset construction* [4]. It is then straightforward to simulate a computation on a given input for this resulting DFA.

This approach works well for classical regular expressions, but it can not support various extensions. Regular expressions with backreferences may be used to describe other than regular languages, even some context-sensitive languages like $\{ww \mid w \in \{a,b\}^+\}$ can be described (see Section 2.3). There is no DFA accepting such non-regular languages and there appears to be no practical way to extend DFA to match rewbr patterns efficiently.

This method is still useful for classical regular expressions. It can also be used in combination with other methods that support backreferences but are therefore potentially slower. First the DFA algorithm is employed to test if the given string matches some pattern approximating[13] the given pattern, and only if it does, the exact and possibly slower NFA match is done.

For instance, the regex pattern $\alpha = x\{\mathtt{a}^*\}\mathtt{b}x$ might be transformed to classical regular expression $\alpha' = \mathtt{a}^*\mathtt{b}\mathtt{a}^*$. The given input would then be first matched against α' using the fast DFA algorithm. Because $\mathcal{L}(\alpha) \subseteq \mathcal{L}(\alpha')$, if the result of DFA algorithm is that the input $w \notin \mathcal{L}(\alpha')$ it must also be

[12] i.e. accepting the same regular language.

[13] The approximating pattern will describe a wider set of strings (i.e. a superset of the original language).

the case that $w \notin \mathcal{L}(\alpha)$. Only if $w \in \mathcal{L}(\alpha')$ must the slower NFA matching algorithm be used to decide if also $w \in \mathcal{L}(\alpha)$. According to [26] this approach is used in some versions of GNU grep.

3.2.2 NFA based with backtracking

This method works by taking the NFA constructed for given regular expression and simulating its computation using *recursive backtracking*. Because many practical regular expression engines (and especially those that support backreferences) are based on this approach, it is examined in detail in Section 3.3. As described in the section, extending engines based on backtracking with backreference support is straightforward.

3.2.3 NFA based without backtracking

There are other ways to simulate a NFA computation than via backtracking. One way is to keep set of active states in each step of computation. The input string is then processed symbol by symbol, updating the set based on all possible transitions. This could be seen as performing the subset construction online and it is also similar to the algorithm used in [23]. This method works well for classical regular expressions, where all parallel branches will differ only in states, and thus keeping only set of active states is sufficient. However, for backreference support, it is necessary to also keep information about capture groups contents. This additional context information can be stored along with the active states. Since it is possible to visit one state with several different capture group contents, there may be many active configurations. For this reason, this method may exhibit high memory usage.

Another approach is to perform an explicit search on the space of automaton configurations, starting with the initial one. Breath-first search or depth-first search algorithms can be used. This way, adding backreference support is again trivial, since we can store the capture group contents inside the configurations. This will also increase the size of the searched space. Since we need to cache the visited configurations, this method may also result in high memory usage. Variant of this technique is used for matching using memory automata and will be described in detail later in Section 4.2.1.

3.3 Recursive backtracking

After obtaining NFA for given regular expression (for instance using the Thompson's construction algorithm), *recursive backtracking* can be used to simulate a NFA computation in the following way. A recursive function, which has an input position and current state as its parameters is used to attempt the match. The first call starts in the initial automaton state and beginning

of input (i.e. initial configuration). For each possible transition from current state, recursive call is made. When the match fails (for instance because the current input symbol does not match), the function simply returns from recursion (backtracks) and another transition is tried instead. This resembles a depth-first search (DFS) on the space of automaton configurations. However, not all visited configurations are necessarily saved, resulting in much lower memory usage than if standard DFS was used. A greatly simplified version of this algorithm is illustrated in the following pseudocode.

Algorithm 1: Naive backtracking

Input : NFA $M = (Q, \Sigma, \delta, q_0, F)$, input string w
Output: True if and only if $w \in \mathcal{L}(M)$

```
1  return Match(q0, 1)
2  Function Match (state q, input index i)
3  |  if q ∈ F ∧ i = |w| then return True
4  |  foreach q' ∈ δ(q, w[i])) do
5  |  |  if Match(q', i + 1) then return True
6  |  end
7  |  foreach q' ∈ δ(q, ε)) do
8  |  |  if Match(q', i) then return True
9  |  end
10 |  return False
```

3.3.1 Preventing infinite loops

What is not handled in the pseudocode of Algorithm 1, is the possibility of entering into an infinite loop. This can be caused by a loop of ε-transitions in automaton, which can be easily generated by patterns such as the one in Example 3.1. An example of such behavior is provided in Example 3.1.

This can be prevented in several ways. For instance, we can *cache* (store) some of the visited configurations. Since this problem is caused by ε-transitions, the configurations resulting from taking ε-transitions can be stored. Additionally, caching is only necesarry for transitions originating from a state with multiple ε. If we already visited the resulting configuration, the transition is not taken again. This approach is better memory-wise than pure depth-first search (see Section 3.2.3), but some unnecessary configurations may still be stored. See also [27] for another approach, which is employed in the implementation of Java programming language.

3.3.2 Backreference support

Extending backtracking with support for backreferences is relatively simple. When we enter the capture group (e.g. the subpattern of the form $x\{\alpha\}$ in

regex), we start recording the input text. Once the subexpression is left, we stop recording. The captured text is stored for each capture group in pattern. These can be stored inside context along with current state and input position (i.e. they would become another parameter in *Match* function in Algorithm 1). Also, since captured texts will always be substrings of the input, it is sufficient to just store pointers into the input string. Then, when the recall occurs in the pattern, the string stored inside the corresponding capture group is matched again. See also [28] for another explanation of this concept.

Other extensions, such as look-ahead, look-behind, or pattern recursion, can also be implemented when using the backtracking approach.

3.3.3 Practical implementations

An early implementation of regular expressions using backtracking is the Henry Spencer's library, available at [29]. It is based on the description in [30]. This library servers as a predecessor and basis for many modern backtracking implementations. For this reason, the term *Spencer-based* engine is sometimes used for such implementations.

A good description of relatively typical backtracking implementation is provided in [27], where the `java.util.regex` package of Java programming language is discussed.

Many other backtracking implementations in use behave similarly, for instance the widely used PCRE library [27].

Note that backtracking does not have to be necessarily implemented using an explicit recursion. Some implementations, such as the ICU Regular expressions that will be discussed in Section 3.5.6, do not use program recursion for maintaining the backtracking state and rather store it on the heap to limit stack usage [31].

3.3.4 Complexity and catastrophic backtracking

The worst-case complexity of matching by backtracking algorithm is exponential [32, 22]. As discussed in Section 2.4, the matching problem for regex is NP-complete. Therefore, unless P=NP, it is not possible to implement a polynomial matching algorithm. However the backtracking algorithm has the exponential worst-case complexity even for regular expressions *without back-references* (classical regular expressions) [22].

Example 3.1. Using the Thompson's construction as described in [4], classical regular expression $\alpha = (\mathtt{a}^*)^*$ would be transformed into the NFA shown in Figure 3.1.

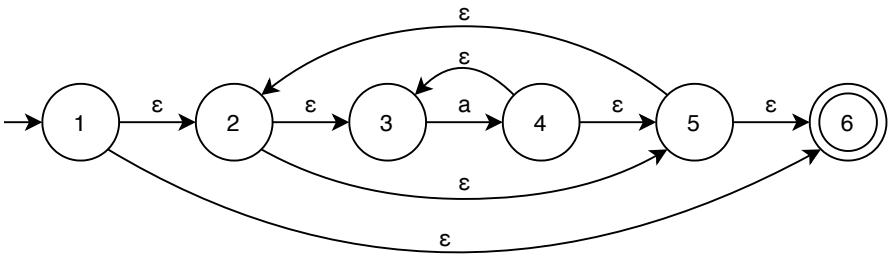


Figure 3.1: NFA resulting from Thompson's incremental construction for regular expression $(\mathtt{a}^*)^*$.

First note, that there is a cycle of ε-transitions between states 2 and 5. If this is not treated using one of the methods from Section 3.3.1, the backtracking algorithm could enter into an infinite loop here.

Now let us consider the computation of this NFA for an input $w = \mathtt{a}^n\mathtt{b}$, for any $n \in \mathbb{N}$. After matching each symbol $\mathtt{a}$, the automaton will be in state 4 and can take one of two ε-transitions from this state: either to state 5 or back to state 3. The backtracking algorithm will use two function calls to try both (see lines 7-9 of Algorithm 1). The first call will eventually return *False*, since $w \notin \mathcal{L}(\alpha)$, and then the second call will be made. However, in each recursive call this decision is made again and two additional recursive calls are made from each. This happens for each of the first to n-th positions in the input. Consequently, 2^n recursive calls will be made before the input w is ultimately rejected. The backtracking algorithm thus exhibits exponential time complexity on such inputs.

One way to prevent this behavior is to cache some visited configurations. As described in Section 3.3.1, this can also be used to prevent infinite loops. If we cache all configurations entered using an ε-transition starting in state with multiple ε-transitions, then configuration $(5, \mathtt{a}^k b)$ will be stored as visited for every $k \in \{n-1, \ldots, 0\}$. When an already visited configuration is entered again, the recursive call may return *False* immediately without making any additional calls, thereby preventing exponential behavior (in this case). As already discussed, the caching of configurations will also result in increased memory usage. ♦

When the exponential matching time occurs, it is also called *catastrophic backtracking*. This behavior can be potentially used for a denial-of-service attacks, when user inputs are not controlled sufficiently. Such attacks, named *regex denial-of-service (ReDos)* are explored in [33], where an algorithm to craft malicious inputs for given regex is presented.

This problem can be averted to some extent. Known inputs that cause catastrophic backtracking can be detected by regular expression engines to

prevent exponential behavior. Modern engines handle several various such special cases. Methods of static analysis to detect vulnerable regular expression patterns are researched in [32] and [27].

Another recent work, which explores ways to prevent this problem is [34]. They propose to add state cache into existing backtracking engines, similar in principle as the visited configurations caching described earlier (see Example 3.1 and Section 3.3.1). However, the author notes that: *"We have not yet considered how to support regexes with super-linear features like backreferences ..."* [34].

3.4 Approaches overview

This section provides an overview of various other methods of regular expression matching, with strong focus on regular expressions with backreferences.

3.4.1 Extending Finite Automata

An approach for implementing regular expressions with backreferences *without* recursive backtracking is presented in [35]. The proposed algorithm is based on the online NFA simulation mentioned in Section 3.2.3. The following subsection briefly describes this method as it was specified in [35]. Then, in the second subsection, the application for multiple capturing groups and various issues are discussed.

3.4.1.1 Method description

The matching is directed by the input and each input symbol is processed at most once. This is achieved by storing a set of *active* states. In each step of computation, the algorithm iterates over all currently active states (stored in the active state set) and performs all possible a-transitions and ε-transitions from each state, where $a \in \Sigma$ is the current input symbol. After that, the algorithm proceeds to the next input symbol until the whole input is processed.

The algorithm described above is basically the already mentioned online NFA simulation. For backreference support, the NFA was extended in the following way. Each backreference in pattern is assigned a unique identifier. This is motivated by matching one input against multiple regular expressions at once in the article, since they focus on networking applications where this is useful. Using identifiers, they can abstract away which backreference belongs to which pattern [35]. This is not as interesting for our problem of matching against single pattern.

Each transition can be marked with up to the number of backreferences different *tags*. A tag is associated with backreference identifier, and it indicates that when it is taken the input symbol should be appended at the end of

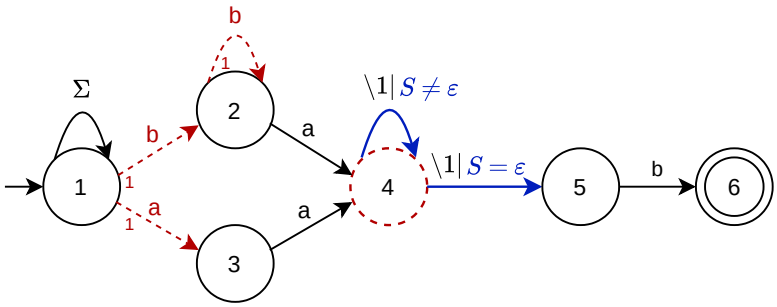


Figure 3.2: Example of an extended NFA for pattern "`.*(b+|a)a\1b`".

the corresponding backreference's recorded contents (string). All transitions to states in the part of automaton constructed from a subexpression inside capturing parentheses (i.e. of the form $x\{\alpha\}$ using[14] regex syntax) will have tags associated with backreferences that recall this subexpression.

Recorded substrings are kept along with active states. For each backreference '$\backslash k$' in the pattern, a set of recorded (previously matched) strings MS_k can be stored for each active state. When a transition $q \xrightarrow{a} p$ is taken, all stored sets are moved from the originating state q to the destination state p. If $a \in \Sigma$, then for all tags k in the taken transition, the matched symbol a is appended at the end of all strings in MS_k.

The backreferences (recalls) are implemented using a special type of state called the *consuming* state. Two special conditional transitions originates from each consuming state $q \in Q$: $q \xrightarrow{S\neq\varepsilon} q$ and $q \xrightarrow{S=\varepsilon} p$ (where $p \in Q, p \neq q$ is next state). When a consuming state q that implements backreference '$\backslash k$' is active, then in a step of the (text-directed) computation the following happens. Suppose that the input symbol processed in current step is $a \in \Sigma$. Then, each string from MS_k that begins with a is shortened by removing a from its beginning (the symbol a is consumed). If the resulting string is empty, then the latter transition $q \xrightarrow{S=\varepsilon} p$ is taken. Otherwise, the former transition back to state q is taken (the string with a removed replaces its original form in MS_k). For nonempty strings in MS_k that do not begin with a, no transition can be taken, meaning that they are removed from the MS_k set of active state q and if no transition to q was taken the state will cease to be active. A special case not explicitly mentioned in the article is when the string captured in MS_k is ε, but this can be treated by taking the transition $q \xrightarrow{S=\varepsilon} p$ (without consuming any input symbol).

[14] This article does not use the regex formalism defined earlier, they use syntax based on PCRE, similar to the one in [13].

Example 3.2. To clarify how the computation of NFA extended in the described way works, the following example is provided. Another two examples can be found in the article [35].

Figure 3.2 shows extended automaton for pattern "`.*(b+|a)a\1b`". Tagged transitions and special states implementing backreferences (recalls) are drawn using dashed arrows, numbers under each transition indicate the backreference tags (here we have only one backreference). The pattern follows PCRE syntax used in the article, '`.`' stands for any symbol and '`|`' denotes alternation. The active states in each step of computation for input "`aabbabb`" will be:

`a`: **1**, **3**($MS_1 = \{\texttt{a}\}$)
`a`: **1**, **3**($MS_1 = \{\texttt{a}\}$), **4**($MS_1 = \{\texttt{a}\}$)
`b`: **1**, **2**($MS_1 = \{\texttt{b}\}$)
`b`: **1**, **2**($MS_1 = \{\texttt{b}, \texttt{bb}\}$)
`a`: **1**, **3**($MS_1 = \{\texttt{a}\}$), **4**($MS_1 = \{\texttt{b}, \texttt{bb}\}$)
`b`: **1**, **2**($MS_1 = \{\texttt{b}\}$), **4**($MS_1 = \{\texttt{b}\}$), **5**
`b`: **1**, **2**($MS_1 = \{\texttt{b}, \texttt{bb}\}$), **5**, **$\underline{6}$**

After processing all input symbols, the computation terminates and input is accepted, because the final state 6 is present in the set of active states. ♦

3.4.1.2 Multiple capturing groups and backreferences

It is not entirely clear how the proposed extended NFA would deal with multiple capturing groups and multiple backreferences in patterns. Both examples from the article have only one capture group and one backreference. It is even claimed in [36] that this Becchi's approach fails when there are multiple backreferences to the same capturing group, but this claim is not explained further. It is stated in [35] that *"each active state can be associated with a set of matched substrings MS_k for each back-reference $\backslash k$"*. The sets MS_k are associated with backreferences not capturing groups, so even if one set is consumed when matching the backreference, subsequent backreferences to the same group would have their own independent sets. As such, this alone does not prevent using multiple backreferences.

However, there is another problem. When an active state has multiple sets MS_k for different backreference numbers, there does not seem to be any information stored on their possible co-occurrences. Not all combinations of captured strings between different backreferences may be possible. This is illustrated in the following example.

Example 3.3. Suppose pattern "`(ab*)(b+)c\1\2`", this is equivalent to pattern $\alpha = x\{\texttt{ab}^*\}y\{\texttt{b}^+\}\texttt{c}xy$ using the regex formalism. The described language is $L = \{ab^ncab^n \mid n \in \mathbb{N}\}$ and it is obvious that $abbcabbb = ab^2cab^3 \notin L$.

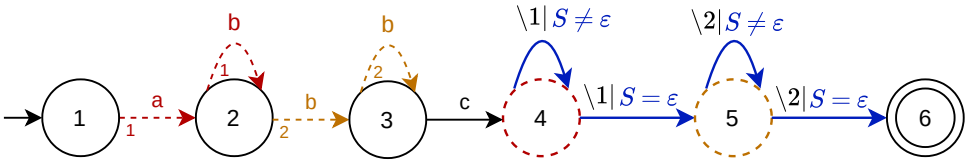


Figure 3.3: Extended NFA for pattern "`(ab*)(b+)c\1\2`" with multiple capturing groups and backreferences.

Figure 3.3 shows extended automaton for this pattern. Given input string "`abbcabbb`", the steps of computation would be (showing active states and their sets MS_k):

`a`: **2**($MS_1 = \{$`a`$\}$)
`b`: **2**($MS_1 = \{$`ab`$\}$), **3**($MS_1 = \{$`a`$\}$, $MS_2 = \{$`b`$\}$)
`b`: **2**($MS_1 = \{$`abb`$\}$), **3**($MS_1 = \{$`a`, `ab`$\}$, $MS_2 = \{$`b`, `bb`$\}$)
`c`: **4**($MS_1 = \{$`a`, `ab`$\}$, $MS_2 = \{$`b`, `bb`$\}$)
`a`: **4**($MS_1 = \{$`b`$\}$, $MS_2 = \{$`b`, `bb`$\}$), **5**($MS_2 = \{$`b`, `bb`$\}$)
`b`: **5**($MS_2 = \{$`b`, `bb`$\}$), **6**
`b`: **5**($MS_2 = \{$`b`$\}$), **6**
`b`: **6**

In the third step (highlighted) of computation, two transitions to state 3 were taken: from state 2 and from state 3 itself. Therefore we have 2 "instances" of active state 3. In the standard NFA, this would not be a problem, but here the two instances have different sets of possible recorded strings: if entered from state 2 we have ($MS_1 = \{$`ab`$\}, MS_2 = \{$`b`$\}$), from state 3 we have ($MS_1 = \{$`a`$\}, MS_2 = \{$`bb`$\}$). We could simply merge the corresponding sets. The result can be seen on the computation above. While "`ab`" is valid contents for backreference "`\1`" and "`bb`" is valid for "`\2`", they can not occur together (for given input). This information is lost by merging the sets MS_k and this leads to the input "`abbcabbb`" being incorrectly accepted.

Solution to this would be to keep multiple instances of an active state (e.g. state 3 in this example) in such cases. ♦

We note that it is not clear whether patterns with multiple (referenced) capturing groups as in the above example were considered at all in the article [35], and if multiple instances of an active state were intended in such cases. It is stated in the article that for input of length m, $O(m^2|Q|)$ is the upper bound on number of all stored strings for one backreference, which seems to imply that multiple instances of an active state were not considered, since each MS_k can contain $O(m^2)$ strings (substrings of input), and there are $|Q|$ states.

As already discussed, if unlimited number of backreferences is allowed, multiple instances of active states may potentially need to be kept as shown in Example 3.3. This can lead to exponentially many such instances. If this was not the case and the "merging" of different instances of an active state in the example worked, the time complexity would be polynomial. This is because there would be at most $O(|Q|)$ active states (at most one instance of each), for each of which there is k sets MS_k, where k is number of backreferences, and each set contains at most $O(m^2)$ strings. Therefore, it would take $O(|Q| \cdot k \cdot m^2)$ to iterate over all of them in each step, and an input of length m could be matched in $O(|Q| \cdot k \cdot m^3)$ using the text-directed algorithm outlined before. The reason why this subsection was included is to clarify that for unbounded number of capture groups, deterministic polynomial time complexity would *not* be achieved. This is expected since, as already established, matching rewbr is NP-complete.

Another edge case not explicitly addressed in the article is occurrence of capture group inside Kleene star or other quantifier expression. Suppose pattern "`(a+|b)+c\1`" (equivalent to pattern $(x\{\mathtt{a}^+ \vee \mathtt{b}\})^+\mathtt{c}x$ using the regex formalism), expected behavior would be that the string captured in the last iteration of subpattern "`a+|b`" is matched by backreference `\1`. For instance, the input `aaabcb` should match. However, as described the automaton would keep appending symbols to strings saved in sets from previous iterations. To prevent this, the transition entering captured subexpression (e.g. $1 \to 2$ in Example 3.2) should first clear any previous MS_k contents (for k corresponding to tags stored in the transition). This would be similar to how memory automata clear previous contents when opening memory (see Chapter 4).

3.4.2 Polynomial time matching for rewbr subclasses

Another noteworthy work is [37], which explores subclasses of patterns whose matching problem has polynomial complexity. They use pattern languages introduced in [17] as a formalism to specify regular expressions with backreferences, but their results can be extended for rewbr in a straightforward fashion [37].

In short, such patterns are (non-empty) strings of terminal symbols and variables and their language is a set of strings obtainable by substituting variables in given pattern with strings of terminals, see [17] or [37] for proper definitions. For instance, the pattern $x_1\mathtt{a}x_2\mathtt{b}x_2$, where $X = \{x_1, x_2\}$ are variables and $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ are terminal symbols, generates (describes) formal language[15] $\{u\mathtt{a}w\mathtt{b}w \mid u, w \in \Sigma^*\}$. This language contains strings such as `aaaba` or `abababba`.

[15]Note that according to the original definition in [17] generated language contains only non-empty strings. The definition in [37] does not seem to have this property.

As mentioned in Section 2.4, Angluin's pattern languages have NP-complete membership problem [17]. Because variables are the source of complexity, the article [37] focuses on *terminal-free* patterns, which are patterns without terminal symbols, i.e. they consist only of variables. For examples in this section, similar notation as in the above articles will be used for such patterns: variables will be denoted using $x_1, x_2, \ldots$ and concatenation will be explicitly marked, such as in: $x_1 \cdot x_2 \cdot x_3$. As an illustration, the pattern $x_1 \cdot x_2 \cdot x_1$ generates language $\{uwu \mid u, w \in \Sigma^*\}$.

The work in [37] was motivated by non-cross patterns explored in [38]. Non-cross patterns have the property that for each variable x_i there is no other variable x_j $(i \neq j)$ in the pattern between first and last occurrence of x_i. For instance, pattern $x_1 \cdot x_2 \cdot x_1$ is not a non-cross pattern since x_2 occurs between first (left-most) and last (right-most) occurrence of x_1. Matching problem of non-cross pattern languages can be solved in polynomial time [38].

An equivalent non-cross property for rewbr would prohibit other backreferences (to different capture groups) between a capture group and its backreference. The rewbr pattern "`(ab*)(b+)c\1\2`" from Example 3.3 would violate such restriction, whereas the one from Example 3.2 would satisfy such non-cross property. When using the matching algorithm from Section 3.4.1 on such non-cross rewbr pattern, it would not be necessary to handle multiple instances of an active state (i.e. the problem outlined in Section 3.4.1.2 would never occur). This is because there must always be at most one non-empty set MS_k at a time, otherwise the pattern would not have the non-cross property. Therefore, such rewbr patterns could be matched by keeping only one MS_k set at a time and the complexity would be polynomial.

The article [37] builds on the concept of non-cross patterns and extends it further by introducing a pattern parameter called the *variable distance*. They describe this parameter as *"the maximum number of different variables separating any two consecutive occurrences of x"*, where x is any variable. For example, the pattern $x_1 \cdot x_2 \cdot x_3 \cdot x_3 \cdot x_1 \cdot x_3$ has variable distance equal to 2, because there are 2 different variables (x_2 and x_3) between the two occurrences of x_1, which is the maximum for this pattern.

They prove that when this parameter is restricted by a constant[16], the resulting matching problem is in P. The non-cross patterns discussed above have variable distance equal to zero. Therefore, variable distance is a complexity parameter, whose restriction induces subclasses of pattern matching problem (and also rewbr matching) with polynomial time complexity. They introduce a computation model called Janus automaton, an extended two-way two-head automaton, which is used to prove their results. [37]

Newer article [2] written recently by Markus Schmid, one of the authors of [37], introduces another complexity parameter for regex called active vari-

[16] i.e. a constant independent from the size of matched pattern.

able degree. This newer complexity parameter is able to describe even larger subclasses of patterns with polynomial matching complexity. Relation between these two parameters will be discussed in Section 4.4.3. The newer article also uses a different computation model – memory automaton, which was first introduced in [1].

Results presented in the recent article [2] using memory automata appear much more promising – not only is the active variable degree more general than the variable distance parameter, but additional different technique for polynomial matching based on a property called *memory determinism* is also presented there. For this reason, this thesis largely focuses on memory automata, they are discussed in separate chapter (see Chapter 4), and the implementation of regex engine that is part of this thesis is also based on them.

This section illustrates that one possible approach to efficient rewbr matching is restricting the patterns by some property. Because the reason for NP-completeness is the presence of backreferences, restrictions pertaining to them seem to be efficient for this purpose. This has a downside that not all regex patterns will satisfy such restrictions, but for substantial subclasses it may be possible to match in polynomial time.

3.4.3 Tagged automata

Ville Laurikari introduced another modification of NFA in [39]. The proposed extension augments automaton transitions with *tags*. Each tag has an associated variable. When a transition with tag is taken, the current input position is saved into the variable.

The main application of such automata with tagged transitions is for submatch addressing and searching in text. So far, we dealt mainly with the matching problem, that is to decide if a given input matches a regex pattern. Submatch addressing allows to report which part of input string got matched by given pattern and its subpatterns (see Section 2.5). This can be implemented using two tagged transitions for each subpattern, whose matched substring should be returned. A transition entering the part of automaton constructed for this subpattern will be tagged and start of matched substring will be saved when this transition is taken. Likewise, when leaving the subpattern another tagged transition is used to save the end of matched substring. This works similarly as tags used in the method described in Section 3.4.1.

In a similar way, tags can be used for searching and also for implementing look-ahead in patterns, as described in [39]. As noted in [35], Laurikari's tagged transitions can not be used directly for implementing backreferences. However, the technique discussed in Section 3.4.1 can be seen as an extension of this approach, where tags with more complex actions (other than just saving position in the input) are used for backreference support. Another different approach based on tagged automata is discussed in the next section.

3.4.4 Tagged NFA with constraints

A way to utilize tagged automata for backreference support was presented in [36]. The basic idea is to first transform the given pattern by replacing every backreference with the subpattern inside the corresponding referenced capturing group. This is similar to the approximation method mentioned in Section 3.2.1, for instance the pattern "`(a+)b\1`" is rewritten to "`(a+)b(a+)`". The transformed pattern is then compiled into a tagged NFA using construction similar to a Thompson's algorithm. As in previous section, the tagged NFA performs submatch addressing and keeps track of the strings matched by particular capturing group. Additionally, constraints are added that compare the corresponding captured strings and ensure that the matched string really is from the language of the original pattern. For instance, in the above example there would be a constraint that the strings matched by the first and the second capturing groups (in the modified pattern) are the same. See the article for detailed description of their proposed matching algorithm. According to their benchmarks, the approach was resistant to tested algorithmic complexity attacks. [36]

3.4.5 Symbolic register automata

Another recent automaton model motivated by dealing with large, possibly even infinite alphabets[17], are *symbolic register automata (SRA)* introduced in [40].

This automaton model has so called *registers* – variables that are able to store symbols, each register stores exactly one symbol. When taking transitions, these register can be written to during automaton computation and later their value can be compared against (recalled).

The article [40] contains examples and describes how regular expressions with backreferences can be implemented using this automaton model. In their benchmarks, an implementation of rewbr matching based on SRA is significantly faster than the Java backtracking implementation.

However, these registers can only hold one symbol and SRAs have finite number of such registers. Therefore, this model has a significant drawback: it can not be used to express regular expressions with capturing groups of unbounded length [40]. For example, capturing subexpression ($\mathtt{a}^+$) can not be implemented, because the matched substring's length is not bounded. This drastically restricts set of rewbr patterns that can be represented by SRA.

[17] Aphabet was defined as a *finite* set of symbols in Chapter 1. However, all the concepts can be also extended for infinite alphabets.

3.4.6 Symbolic regex matcher

An alternative method of regular expression matching based on derivatives was developed in [41]. Derivatives of regular expressions are an established concept, see [25]. The algorithm in [41] works with derivatives of extended variant of regular expressions. Their patterns support both character classes and counting constraints (see Section 2.6), as well as additional operations like intersection and complement. Their method allows to not only decide if an input matches the pattern, but also to find all the matches (they call this task *match generation*).

The algorithm is non-backtracking and has linear time matching complexity in the length of the input. However, backreferences in patterns are not supported. An implementation in C# language is provided. [41]

3.4.7 Pattern optimizations

Another way to speed up the matching process is by optimizing the regular expression pattern itself. Optimizations are usually done on some internal representation of the given pattern, for instance on abstract syntax tree, or even on the resulting automaton structure, which will then be used by the matching algorithm.

For DFA based regular expression engines discussed in Section 3.2.1, the automaton can be minimized using e.g. Hopcroft's minimization algorithm [42], which transforms the DFA into an equivalent DFA that has a minimal number of states. This is, however, not generally usable for implementations based on NFA, such as backtracking implementations, where the NFA is also extended to support additional features like backreferences. Still there are several techniques to help the engine by transforming the pattern to equivalent one that will lead to faster matching, many such transformations are described in [10].

For instance, it may help to expose literal text by factoring out common prefixes in alternation expressions. For instance, the pattern "`abcd|abef`" can be rewritten as "`ab(?:cd|ef)`", which may lead to earlier recognition of mismatch and also help the engine to perform other literal-text optimizations [10]. The same can also be done for common suffices, such as transforming "`xabc|yabc`" into "(?:`x|y`)abc".

Because alternation in patterns is expensive, as it may cause backtracking for the alternate branches, it may be helpful to convert alternation between characters into character classes. For example, pattern "`a|b|c`" can be converted into the equivalent "`[abc]`", which can be matched in one step of the regular expression engine.

Optimizations on the level of patterns can be done by the engine, as well as manually by the user. However, some implementations contain advanced opti-

mization logic and outright discourage hand-optimizing the regular expression by user, for example as in the case of the Hyperscan library [43].

3.4.8 Virtual machine approach and JIT compilation

A different approach to regular expression matching is to compile given pattern into some form of machine code. An input can then be matched against the pattern by executing this machine code. Similar approach was taken in the original Henry Spencer's library [29] mentioned in Section 3.3.3.

However, instead of compiling into machine code for an existing processor, the pattern can instead be compiled into instructions for some *virtual machine (VM)*, the result is also called a *byte code*. A regular expression virtual machine can be defined for this purpose, with instructions for matching symbols (comparing against given character and advancing in input), returning successful match and so on.

This family of approaches is described in detail in [26]. The virtual machines can be implemented using recursive backtracking, but non-recursive implementation is also possible. If the alternation construct is implemented using some sort of *split* instruction, which creates a separate thread of execution, as in the referenced article, then parallelization of such implementation can be done intuitively. Some of the threads of execution can be run in separate OS threads.

This approach can also support backreferences as outlined in [26]. It can be done e.g. by introducing a *save* instruction and saving the captured strings in context (current thread state).

The VM approach was also used for regular expressions in text editor Sam developed by Rob Pike [44].

The compilation into instructions during execution (i.e. at run time instead of before execution) is called *just-in-time (JIT) compilation*. In context of regular expression engines, the pattern may be compiled directly into target platform machine code, as in the Henry Spencer's library, or into byte code for some virtual machine, either a special VM for regular expressions or an already existing VM, such as the Java virtual machine (JVM).

More complex variants are also possible, such as compiling the pattern into abstract syntax tree (AST), or some other internal representation, and then using it to generate the machine code. This scheme was used in [45] to implement JIT compilation support for the PCRE library. Figure 3.4 shows overview of this implementations' architecture. The PCRE byte code generated for given pattern is first transformed into an intermediate representation used by SLJIT compiler, which then generates machine code for target platform from it, see [45] for detailed description. According to benchmarks in

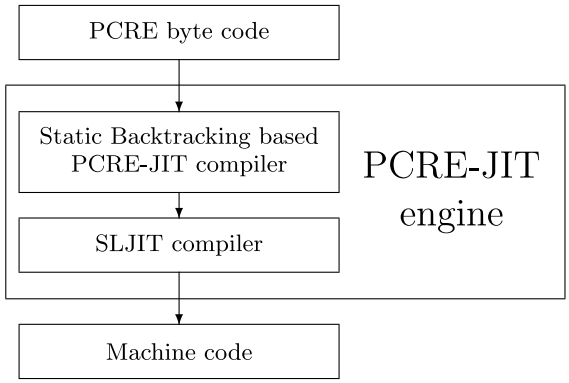


Figure 3.4: Overview of PCRE-JIT engine [45]

the referenced article, using JIT compilation resulted in significant speed up for the PCRE engine.

3.5 Overview of major implementations

This section provides basic overview of practical regular expression engines.

3.5.1 Perl Compatible Regular Expressions

The *Perl Compatible Regular Expressions (PCRE)* [46] is an open source regular expression library written in C language, it is available under the BSD license. A native wrapper for C++ programming language is also provided. The PCRE library has two major versions, the first PCRE version was originally published in 1997, while the more recent PCRE2 was first released in 2015. The syntax follows the Perl regular expression syntax (specifically Perl 5) and the library provides wrapper functions following the POSIX regular expression standard (see Section 2.7). [46]

The PCRE library supports a wide array of features including backreferences. Two matching algorithms are available. First is a backtracking NFA based algorithm with support for backreferences. The second one is DFA based algorithm (see Section 3.2.1), which does not support backreferences and other extra features, and is not Perl-compatible [47]. The backtracking algorithm also supports just-in-time optimization described in Section 3.4.8 [47, 45].

The PCRE library is one of the most widely used regular expressions implementations. It is used for regular expression support in PHP, R, Delphi and Xojo (formerly REALbasic) programming languages [48]. It is also incorporated into other applications such as the Apache HTTP server and Nginx web server.

3.5.2 Java

Regular expression matching in Java programming language is provided in the `java.util.regex` package. It is implemented using the backtracking algorithm and does support backreferences and other additional features. It is a fairly typical backtracking implementation, good description of the package implementation is provided in [27].

3.5.3 JavaScript

Due to its widespread use, an important dialect of practical regular expressions is the variant used in JavaScript programming language. It is not a single implementation, the JavaScript language, including the syntax and semantics of its regular expressions, is standardized in the ECMAScript specification [49]. Significant number of different JavaScript implementations (engines) is in use today, largely as part of web browsers. Naturally, the specific behavior between certain implementations may vary.

The specification of regular expression in the ECMAScript standard is based on Perl 5 syntax, similarly as in the case of PCRE library. The standard does contain backreferences in patterns and conforming implementations may additionally support other extensions [49].

Google's V8 JavaScript engine, which is used for instance in Google Chrome web browser, employs a regular expression implementation called Irregexp [50]. It is a backtracking implementation, which first converts given pattern into an intermediate automaton representation, then performs optimizations and generates native machine code [50].

3.5.4 Oniguruma

Another open source library written in C is Oniguruma, its source code is available at [51]. It also supports backreferences. The matching is implemented using VM based approach similar to the one discussed in Section 3.4.8, the pattern is first transformed into an abstract syntax tree and then compiled into instructions for Oniguruma virtual machine [52]. Oniguruma is used by Ruby programming language, as well as in some applications, such as Atom text editor.

3.5.5 Boost regex

Another open source implementation [53] of regular expressions is part of the Boost C++ library. It uses backtracking and supports backreferences. This implementation and its semantics is explored in [54].

The Boost regex module also heavily influenced the proposal to add regular expression support into the C++ standard [55]. Since C++11, regular expressions are part of the C++ standard [56].

3.5.6 ICU Regular Expressions

An implementation of regular expressions is part of the ICU – International Components for Unicode project [31]. Its C++ API is inspired by the interface of the `java.util.regex` package of the Java language. It is a bactracking implementation, however it uses heap for storing its state during matching, and does not use recursion to perform the backtracking. This is done to limit stack usage, which could be a problem on complex patterns. The supported syntax and pattern semantics are based on regular expressions in Perl. [31]

3.5.7 GNU C library and POSIX

As described in Section 2.7, regular expressions are part of the POSIX standard. The GNU C library (glibc) [57] includes implementation of regular expressions, it supports both POSIX basic and extended regular expression syntax. The implementation is based on backtracking and it supports backreferences. The library is distributed under GNU General Public License.

3.5.8 Hyperscan

Another regular expression matcher called Hyperscan was introduced in [58], where the internal design and implementation details are described. The implementation is available under the BSD license at [59], it has a C API. This engine is not based on recursive backtracking, and it does *not* support backreferences, and various other features are also missing [43].

3.5.9 TRE

Another open source library is TRE [60]. It supports backreferences and attempts to closely follow the POSIX standard. It also supports approximate pattern matching (it uses Levenshtein distance as a measure), which is a feature not commonly found in other regular expression engines.

3.5.10 RE2

Last library mentioned in this overview is Google's RE2 regular expression library [61]. The engine is non-backtracking and does *not* support backreferences. This library is also open source, and it is written in C++, but wrappers for other languages are provided.

Chapter 4

Memory automata

In this chapter, the memory automaton model, which accepts regex languages, is introduced. First a formal definition is given, then the properties of memory automata and their relation to regex languages are explored. Additionally, matching algorithms based on memory automata are discussed along with recently published methods for polynomial matching.

4.1 Memory automaton model

Memory automaton is a model of computation that characterizes the regex language class (see Sections 2.2.3 and 2.3). This model was first introduced in [1]. Later works [2] and [3] also use this model and develop new approaches based on it.

As the regex patterns (introduced in Section 2.2.3) are extension of classical regular expressions, the memory automaton model is an extension of nondeterministic finite automaton. Intuitively, memory automaton is an NFA augmented with finite number of *memories* that can record and store substrings of processed input. The strings stored in memories can later be *recalled* by consuming (matching) the corresponding string from the input again.

The model will now be defined formally. The below definition follows the one in [2], but memory automata are defined very similarly in [1], albeit with some minor differences.

Definition 4.1 (memory automaton)**.** Let $\Gamma_k = \{o(x), c(x) \mid x \in [k]\}$, for $k \in \mathbb{N}$. Additionally, for an alphabet Σ, let[18] $\Sigma_k = \Sigma \cup [k]$ and $\Sigma_{\varepsilon,k} = \Sigma_k \cup \{\varepsilon\}$.

For $k \in \mathbb{N}$, a *k-memory automaton* (denoted MFA(k)) is syntactically an NFA $(Q, \Delta, \delta, q_0, F)$, where $\Delta = \Sigma_{\varepsilon,k} \cup \Gamma_k$. Semantics are defined in the following way. Configuration of MFA(k) is a (k+2)-tuple $(q, w, (u_1, r_1), \ldots, (u_k, r_k))$,

[18] Here it is assumed that $\Sigma \cap [k] = \emptyset$.

where $q \in Q$ is the current state, w is the remaining input, and (u_i, r_i) is the *configuration of memory* i, for $i \in [k]$. Configuration (u_i, r_i) of memory i consists of *memory content* u_i, which is a substring of the input, and *memory status* $r_i \in \{\mathtt{O}, \mathtt{C}\}$. The *initial* configuration of M on input $w \in \Sigma^*$ is $(q_0, w, (\varepsilon, \mathtt{C}), \ldots, (\varepsilon, \mathtt{C}))$. Configuration $(q, \varepsilon, (u_1, r_1), \ldots, (u_k, r_k))$ is *accepting* if $q \in F$. If the number of memories is not important, memory automaton may be referred to simply as MFA.

The transition relation $\vdash_M$ is a relation on the configurations of MFA(k). For configurations c and c', $c \vdash_M c'$ iff one of the following two conditions hold:

1. $c = (q, vw, (u_1, r_1), \ldots, (u_k, r_k))$ and $c' = (p, w, (u'_1, r_1), \ldots, (u'_k, r_k))$, where:
   a) $p \in \delta(q, x)$ with either $(x \in \Sigma_\varepsilon \wedge v = x)$ or $(x \in [k], r_x = \mathtt{C} \wedge v = u_x)$,
   b) and, for every $l \in [k], r_l = \mathtt{O}$ implies $u'_l = u_l v$, and $r_l = \mathtt{C}$ implies $u'_l = u_l$.
2. $c = (q, w, (u_1, r_1), \ldots, (u_k, r_k))$ and $c' = (p, w, (u_1, r_1), \ldots, (u'_l, r'_l), \ldots, (u_k, r_k))$, where $p \in \delta(q, x)$ with either
   a) $x = o(l), r'_l = \mathtt{O} \wedge u'_l = \varepsilon$,
   b) or $x = c(l), r'_l = \mathtt{C} \wedge u'_l = u_l$.

The language accepted by a MFA is defined in an analogous way to the definition for NFA. An input $w \in \Sigma^*$ is accepted by memory automaton M if $(q_0, w, (\varepsilon, \mathtt{C}), \ldots, (\varepsilon, \mathtt{C})) \vdash^*_M (q, \varepsilon, (u_1, r_1), \ldots, (u_k, r_k))$, where $q \in F$. The language accepted by M is the set of all strings accepted by M and will be denoted $\mathcal{L}(M)$. A sequence of configurations $c_1 \vdash_M c_2 \vdash_M \ldots \vdash_M c_n$ is called *computation of* M, if c_1 is initial configuration on w, it is called *computation of* M *on input* w.

Similarly as in the case of NFA, for $x \in \Delta$, transitions $p \in \delta(q, x)$ are called x-transitions. As with NFA transitions, $p \in \delta(q, x)$ can also be written as $q \xrightarrow{x}_\delta p$ (or $q \xrightarrow{x} p$ when the context is clear). If $x \in [k]$, such x-transitions are called *memory recall transitions*. △

As seen from Definition 4.1, MFA(k) is NFA extended with k memories, each of which has a contents (saved string) and a status O or C. Memories with status O will be referred to as *open* memories and those with status C as *closed* memories.

Apart from the classical x-transitions for $x \in \Sigma_\varepsilon$, which are modified to save processed input into open memories, there are additional special transitions for manipulating with memories. In the above definition of relation $\vdash_M$, rule 1 handles x-transitions for $x \in \Sigma_\varepsilon \cup [k]$. The case 1a states what string is

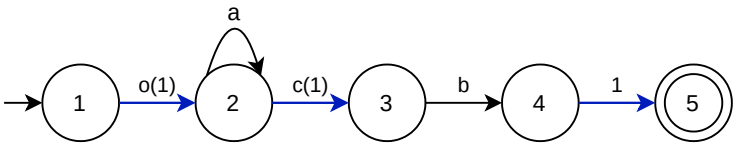


Figure 4.1: MFA(1) accepting language $\{a^n b a^n \mid n \in \mathbb{N}_0\}$

going to be consumed (matched) from input. For the "classical" x-transitions with $x \in \Sigma_\varepsilon$, the symbol x is consumed (or nothing in case of ε). For memory recall transitions with symbol $x \in [k]$, the string stored in memory x is consumed (matched) from the input, the memory contents is not changed meaning that the string stored in memory may be recalled multiple times. Rule in case `1b` states that all open memories will have the consumed string appended at the end of its content. Contents of closed memories will not change.

Additionally, rule `2` deals with x-transitions for $x \in \Gamma_k$ (that is $o(l)$ and $c(l)$ transitions, where $l \in [k]$). Transition $o(l)$ opens the memory l, it erases its previously stored content and sets its memory status to `O` (opened). Transition $c(l)$ closes the memory l, memory status is set to `C` (closed), but memory content is not modified.

Example 4.1. Figure 4.1 shows an 1-memory automaton (MFA(1)) accepting the language $L_1 = \{a^n b a^n \mid n \in \mathbb{N}_0\}$ from Example 2.1. The transition $1 \xrightarrow{o(1)} 2$ opens memory 1 (and sets its content to ε), then the string a^k matched by repeatedly taking transition $2 \xrightarrow{a} 2$ is stored into the memory 1. When transition $2 \xrightarrow{c(1)} 3$ is taken, memory 1 is closed. After that symbol `b` is matched and then the string stored in memory 1 is recalled when taking transition $4 \xrightarrow{1} 5$.

To illustrate how the memory automaton operates, a computation of this MFA for input `aabaa` is given below:

$(1, \texttt{aabaa}, (\varepsilon, \texttt{C})) \vdash (2, \texttt{aabaa}, (\varepsilon, \texttt{O})) \vdash (2, \texttt{abaa}, (\texttt{a}, \texttt{O})) \vdash (2, \texttt{baa}, (\texttt{aa}, \texttt{O})) \vdash (3, \texttt{baa}, (\texttt{aa}, \texttt{C})) \vdash (4, \texttt{aa}, (\texttt{aa}, \texttt{C})) \vdash (5, \varepsilon, (\texttt{aa}, \texttt{C})) \vdash \texttt{accept}$

Note that the automaton contains nondeterminism. When in configuration $(2, \texttt{aabaa}, (\varepsilon, \texttt{O}))$, both transition $2 \xrightarrow{a} 2$ and transition $2 \xrightarrow{c(1)} 3$ could have been taken (similar situation also happens in the next configuration). For more complicated automata this nondeterminism may result in many possible computations. See also [2] for another more complicated example. ♦

The memory automaton formalism has the advantage that a MFA can also be interpreted as a NFA accepting language extended with meta-symbols. Both automaton models are used to explore properties of regex languages

and develop techniques for polynomial regex matching in [2]. Since memory automata accept exactly the class of regex languages (see Section 4.3), they can also be used as an alternative definition of the regex languages and are a useful tool for proving various properties of that language class.

Operation of MFA is somewhat similar to the other NFA extension [35] described in Section 3.4.1. However, while the automaton from [35] was outlined informally by describing the matching algorithm, memory automaton computation is defined precisely using transition relation on MFA configurations. Additionally, unlike the former extension memory automata do not suffer from the problems explored in Section 3.4.1.2.

4.2 Memory automata for regex patterns

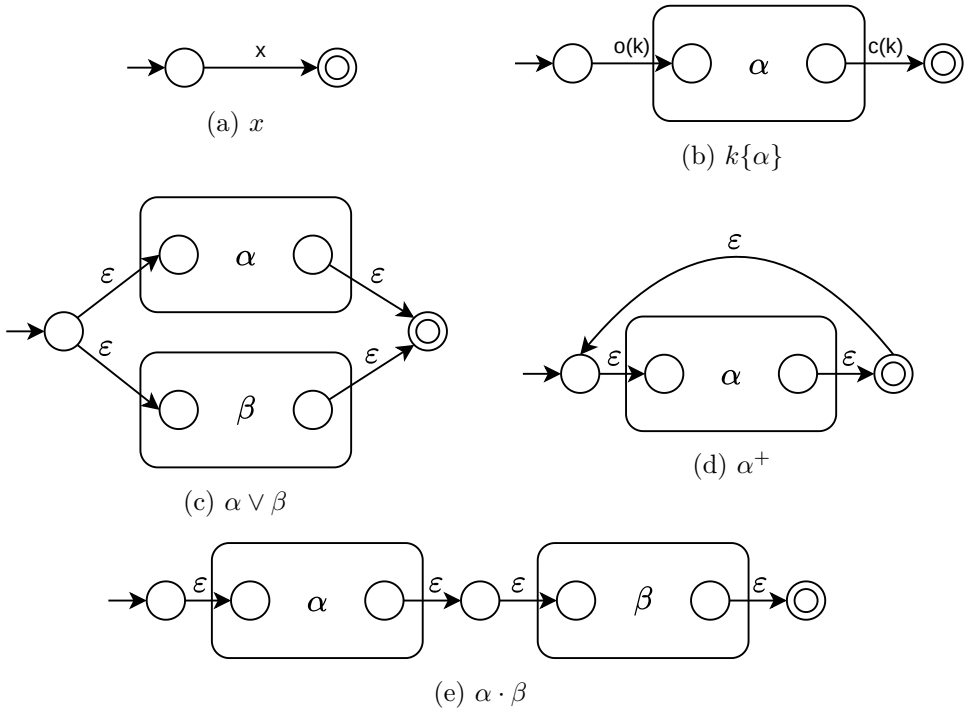
A method to construct memory automaton for any regex pattern is described in [2]. It is an incremental construction similar to Thompson's construction method. Their algorithm uses syntax tree of regex pattern as input and produces MFA by recursively constructing states and transitions for every node of the tree. Syntax tree of regex can be defined naturally according to rules in Definition 2.1. For instance, pattern of the form $\alpha \vee \beta$ (from rule `2b`) would have syntax tree with node $\vee$ as its root and roots of syntax trees of the subpatterns α, β as its children. See [2] for detailed definition of regex syntax tree and precise description of the construction algorithm.

Figure 4.2 illustrates how the memory automaton construction works. Each sub-figure shows structure of automaton for one of the rules from Definition 2.1. Similarly as in the Thompson's construction algorithm, memory automata ("fragments") constructed for subpatterns of a pattern are recursively combined into larger automaton describing the entire pattern. When combining, transitions incoming into child fragments go into their initial states and outgoing edges originate from their final states. As seen in the figure, the resulting automaton will always have exactly one initial and one final state.

Memory automaton constructed in this way will also have a constant number of outgoing transitions for each state: there are always at most two outgoing transitions. Furthermore, all outgoing transitions will be labeled with the same symbol.

Because the algorithm uses syntax tree as an input, if the pattern is given in form of a string (as will likely be the case in practice), it must be first converted into its syntax tree. Since the language of regex pattern notation is context-free[19], this can be done using one of the methods of syntax analysis described in [62].

[19]The rules in Definition 2.1 could be used to create a context-free grammar for regex pattern notation.



(a) x

(b) $k\{\alpha\}$

(c) $\alpha \vee \beta$

(d) α^+

(e) $\alpha \cdot \beta$

Figure 4.2: Illustration of regex to memory automaton conversion

According to the construction rules, a constant number of states is added for each type of syntax tree node. It thus holds that $|Q| = O(|\alpha|)$. This property can also be preserved if the input is given as a string instead of syntax tree.

No rule for α^* is provided, but following Definition 2.1, α^* can be constructed by treating it as $\alpha^+ \vee \varepsilon$. Additionally, the pattern $\emptyset$ can be characterized by an automaton with two states, one initial and the other final, and no transitions.

4.2.1 Matching regex using memory automata

Using the construction described in previous section, it is possible to construct for any given regex pattern α a memory automaton M, such that $\mathcal{L}(M) = \mathcal{L}(\alpha)$. Therefore, the regex matching problem (i.e. to decide whether $w \in \mathcal{L}(\alpha)$) can be solved by first constructing the corresponding MFA M and then deciding $w \in \mathcal{L}(M)$. To decide $w \in \mathcal{L}(M)$, a breadth-first search (BFS) can be run on the set of all possible configurations (of the MFA M), starting in the initial configuration [2].

The construction algorithm runs in time[20] $O(|\alpha|)$ [2]. Additionally, a proof is provided in [2] that if an MFA(k) $M = (Q, \Sigma, \delta, q_0, F)$ satisfies $|\delta| = O(|Q|)$, the BFS search on configurations takes time $|Q||w|^{O(k)}$.

The $|\delta| = O(|Q|)$ constraint is always satisfied for an automaton constructed for regex in the described way, since there is a constant number of transitions for each state. It can also be seen from the construction rules that the resulting MFA will have one memory for each variable (capture group) in pattern. Therefore, using this algorithm, regex pattern α with k variables can be matched against an input w in time $O(|\alpha|) \cdot |w|^{O(k)}$. If the number of variables k is bounded by a constant, the matching time complexity is polynomial. Otherwise, there can be up to $O(|\alpha|)$ variables, in which case the time complexity would be $O(|\alpha|) \cdot |w|^{O(|\alpha|)}$ and therefore exponential with pattern length.

Two main approaches for efficient regex matching were introduced in [2]. In both cases some restricting property is formulated for regex patterns and a polynomial time matching algorithm is provided for the resulting regex subclass. An example of efficiently matchable subclass of regex are patterns with number of variables (backreferences) bounded by a constant as discussed earlier. Similar restriction yielding larger subclass is to require only the number of memories to be bounded by a constant. So far, the presented construction algorithm always used one memory for each variable, but it is possible to match regex using memory automaton with less memories than pattern variables. This is basis for the first approach from [2] that uses property called active variable degree, it is the topic of Section 4.4 below.

The second approach does not limit number of memories but restricts how the memories are used, it is explored in Section 4.6. A similar property inducing much more restricted subclass of regex was established in [3] and is the focus of Section 4.5.

4.3 Properties of memory automata and relation to regex languages

The class of languages accepted by k-memory automata is equal to the class of regex languages [1]. Regex patterns and memory automata thus have equal expressive power and MFA can be used as an alternative way to define the class of regex languages. A way to convert regex into an equivalent MFA was discussed in Section 4.2. For the opposite direction, a memory automaton can also be transformed into an equivalent regex, see [1].

Now it will be shown that computation of a k-memory automaton can be simulated using linear-bounded automaton (i.e. on nondeterministic Turing

[20] They assume syntax tree as an input but regex pattern can be converted into its syntax tree in linear time using e.g. LR parsing [62].

machine in linear space). This also implies the property of regex languages being (proper) subset of context-sensitive languages discussed in Section 2.3. k-memory automata can be interpreted as NFA with special transitions (labeled with meta-symbols) and additional k memories. Each of the k memories has status and content. Statuses are boolean variables and storing them for k memories requires $O(k)$ space. Memory content can be stored by either storing the string, which is substring of the input w and so will have length at most $n = |w|$, or by keeping start and end pointers into the input string. The former way requires $O(k \cdot n)$ memory, the latter only $O(k)$, but since the given input string is also stored, both alternatives have linear memory complexity.

For the computation, MFA(k) can be interpreted as a NFA accepting special meta-symbols. When $o(x), c(x)$ or x symbol (for $x \in [k]$) is consumed, the corresponding memory operation is performed. The computation runs as described in Section 4.1, by simulating the computation on nondeterministic Turing machine all the parallel computational branches (created by nondeterministic choices) can be handled simultaneously. It can also be seen on an intuitive level, that the computation of k-memory automaton model could be simulated on $(k+1)$-tape linear bounded automaton with input on one tape and the other k tapes serving as memories.

4.4 Active variable degree

A property named active variable degree was introduced in [2]. Subclass of regex obtained by restricting this property and an algorithm for efficient matching is also presented in the article.

Basic idea of this approach is reusing a memory for multiple variables in regex patterns. For instance, the pattern $\alpha = x_1\{\mathtt{a}^*\}\mathtt{c}\,x_1\mathtt{c}\,x_2\{\mathtt{b}^*\}\mathtt{c}\,x_2$, which represents language $\mathcal{L}(\alpha) = \{a^n c a^n c b^m c b^m \mid n, m \in \mathbb{N}_0\}$, contains two variables. If the construction algorithm discussed in Section 4.2 was used, the resulting memory automaton would also have two memories. However, as illustrated in Figure 4.3, it is possible to construct memory automaton with just one memory (i.e. a MFA(1)) accepting the same language. Because variables in this pattern are "independent", the single memory can be used first to save a^n and then recall it, and later to do the same for b^m.

By contrast, in case of regex pattern $\beta = x_1\{\mathtt{a}^*\}\mathtt{c}\,x_2\{\mathtt{b}^*\}\mathtt{c}\,x_1x_2$ the memory used to store a^n can not be reused in the same way for storing b^m, because value of the first variable is still needed later (it is recalled after $b^m c$ is matched). Here the variables are not "independent" in the sense they were in pattern α. Intuitively, this corresponds to non-cross patterns mentioned in Section 3.4.2, but as will be shown later, active variable degree is more general, i.e. it allows significantly larger subclass of regex to be matched efficiently.

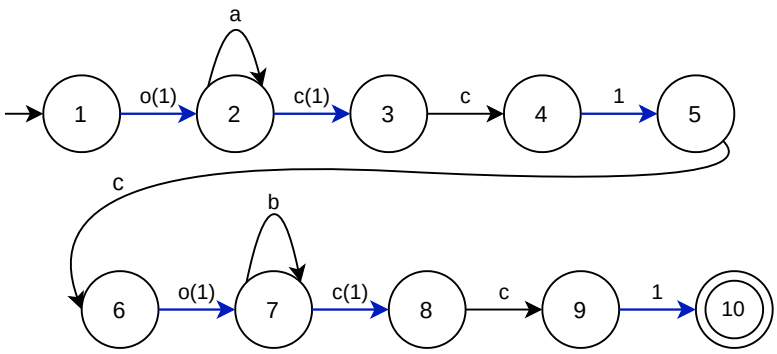


Figure 4.3: MFA(1) accepting language $\{a^n c a^n c b^m c b^m \mid n, m \in \mathbb{N}_0\}$

A different example where less memories than variables are required is pattern $\gamma = (x_1\{\mathtt{a}^*\} \vee x_2\{\mathtt{b}^*\})x_1x_2$. Either variable x_1 or variable x_2 is defined, but never both at the same time. Only single memory is thus needed, one of the variables will be undefined (and thus count as ε when using ε-semantics).

To formalize this behavior and what it means for variables to be "independent", the active variable degree property was introduced in [2]. Intuitively, a set of *active* variables is defined for each state of memory automaton, containing all variables that can be defined when in this state and at the same time are also possibly recalled later. Value of variables that are not active in current state is not important for further computation. Active variable degree is then defined to be the maximal active variable set size, which is equal to the minimal number of required automaton memories to handle active variables during computation. These concepts are established formally in Definition 4.2, which follows closely the one from [2].

Definition 4.2 (active variable set, active variable degree)**.** Let $\alpha \in \mathrm{RX}_{\Sigma,X}$ and $\mathcal{M}(\alpha)$ the MFA($|X|$) constructed for pattern α (i.e. $\mathcal{L}(\mathcal{M}(\alpha)) = \mathcal{L}(\alpha)$) with its states denoted using Q. Additionally, $\mathcal{R}(\alpha)$ denotes the same automaton interpreted as NFA.

Relations $\triangleright_{\mathrm{def}} \subseteq X \times Q$ and $\triangleright_{\mathrm{call}} \subseteq Q \times X$ defined as follows:

- For $x \in X, q \in Q$: $x \triangleright_{\mathrm{def}} q \Leftrightarrow \mathcal{R}(\alpha)$ can reach q by reading string w with[21] $|w|_{o(x)} \geq 1$.
- For $q \in Q, x \in X$: $q \triangleright_{\mathrm{call}} x \Leftrightarrow$ starting in q, $\mathcal{R}(\alpha)$ can read string wx with $|w|_{o(x)} = 0$.

[21] As defined earlier, $|w|_a$ stands for number of occurrences of symbol a inside string w, therefore the condition $|w|_{o(x)} \geq 1$ requires that at least one $o(x)$ was in the string read by $\mathcal{R}(\alpha)$, i.e. at least one $o(x)$-transition was taken.

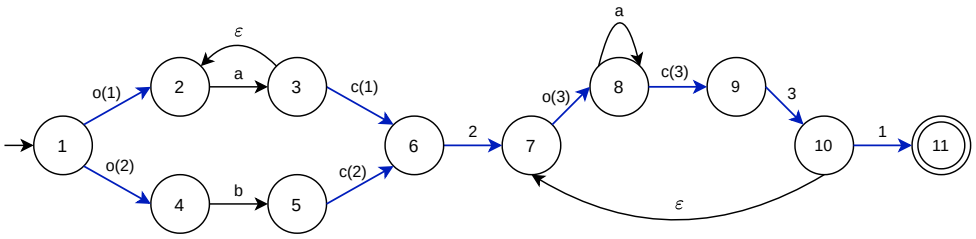


Figure 4.4: MFA(3) for regex pattern $\alpha = (x_1\{\mathtt{a}^+\} \vee x_2\{\mathtt{b}\})x_2(x_3\{\mathtt{a}^*\}x_3)^+x_1$

For every state $q \in Q$ the *set of active variables (for q)*, denoted $\mathtt{avs}(q)$, is defined as $\mathtt{avs}(q) = \{x \in X \mid x \triangleright_{\text{def}} q \wedge q \triangleright_{\text{call}} x\}$.

Active variable degree of α, denoted $\mathtt{avd}(\alpha)$, is then defined as maximum from active variable sets' sizes: $\mathtt{avd}(\alpha) = max\{|\mathtt{avs}(q)| \mid q \in Q$ and a transition labeled $o(l)$ leads into q, where $l \in X\}$.

In the original definition from [2], the above condition for q is formulated using syntax tree, but this is an equivalent property because of how the automaton is constructed. This condition only states that the only interesting states for computing active variables are the ones entered using $o(x)$ transition (because of how the relation $\triangleright_{\text{def}}$ is defined).

Additionally, for every $k \in \mathbb{N}$, the set of *regex with active variable degree at most k* is defined as $\mathrm{RX}^{\mathtt{avd}\leq k}_{\Sigma,X} = \{\alpha \in \mathrm{RX}_{\Sigma,X} \mid \mathtt{avd}(\alpha) \leq k\}$. △

The relation $\triangleright_{\text{def}}$ from Definition 4.2 formalizes which variables[22] may have recorded value in each state. The relation $\triangleright_{\text{call}}$ indicates if a memory can be recalled later (without its value being redefined) when starting in given state. If a memory is both possibly defined and also recalled later from some state, its value must be kept when in this state and it will be in active variable set.

Example 4.2. Figure 4.4 depicts a MFA(3) constructed for regex pattern $\alpha = (x_1\{\mathtt{a}^+\} \vee x_2\{\mathtt{b}\})x_2(x_3\{\mathtt{a}^*\}x_3)^+x_1$. Note that the automaton is slightly different from what the result of construction algorithm would be. To maintain brevity, unnecessary ε-transitions were removed, as they are not important for this example.

The following table shows relations and active variable sets for each state. The $\triangleright_{\text{def}}$ row contains for each state $q \in Q$ all the variables (memories) $x \in X$ for which $x \triangleright_{\text{def}} q$. Similarly, the row $\triangleright_{\text{call}}$ lists all x, s.t. $q \triangleright_{\text{call}} x$. The last row then contains each state's $\mathtt{avs}(q)$.

[22] In the Definition 4.2, the number of variables and memories is the same and so the terms variable and memory are used interchangeably, it is assumed that $X = \{1, 2, \ldots, k\}$, where $k \in \mathbb{N}$, so $|X| = k$.

state	1	2	3	4	5	6	7	8	9	10	11
$\triangleright_{\mathbf{def}}$	$\emptyset$	1	1	2	2	1,2	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3
$\triangleright_{\mathbf{call}}$	1,2	1,2	1,2	1,2	1,2	1,2	1	1,3	1,3	1	$\emptyset$
`avs`	$\emptyset$	1	1	2	2	1,2	1	1,3	1,3	1	$\emptyset$

The active variable degree here is 2, all the active variable sets have at most 2 elements. Therefore, 2 memories are sufficient to match this regex, the procedure to do so is discussed in Section 4.4.1.

A shortcoming of the `avd` property can be seen in this example. Variables 1 and 2 can be defined in state 6 and, because they both can also be recalled from this state, the active variable set of state 6 contains both of them. This also means that even without variable 3 in pattern, the `avd` would still be 2. However, it is impossible to reach state 6 with variable 1 and variable 2 defined at the same time. During the MFA computation, either the transition to state 2 or the one to state 4 is taken from initial state 1, which determines whether variable 1 or 2 will be defined, but they can not be defined simultaneously. Therefore, if the variable 3 would not be in pattern, only at most one variable could be active at one time during computation, but the active variable property does not capture this. An extension of `avd` that is able to recognize occurrence of such behavior in patterns is discussed later in Section 4.4.2, unfortunately computing that extended property is intractable. ♦

It is shown in [2] that the relations $\triangleright_{\text{def}}$ and $\triangleright_{\text{call}}$ (and therefore also the active variable sets) can be computed in time $O(|X||\alpha|^2)$, the active variable degree $\texttt{avd}(\alpha)$ can then be computed by taking maximum from sizes of active variable sets. In the proof, a way to check if two elements are in relation is presented for both relations. In both cases it is based on using cross-product of some nondeterministic automata and checking reachability between configurations, see [2] for details. Although this is useful for formal proof, doing this construction explicitly and checking relation for every state-variable pair in this way would be needlessly complicated in practice. An alternative algorithm for computation of active variable sets based on BFS traversal of MFA states is presented later in this thesis in Section 5.1.3.

4.4.1 Matching regex with bounded active variable degree

The matching problem for regex with active variable degree at most $k \in \mathbb{N}$ can be solved in polynomial time [2]. The referenced article provides a high level description of how a MFA(k) M constructed for regex pattern α can be simulated using MFA($\texttt{avd}(\alpha)$) M'. Intuitively, the k memories of M that are active are mapped into the memories of M'. When a memory operation is to be performed on an old memory $l \in [k]$, it is done on the memory to which memory l is currently mapped. When a memory is not active (not

in the active variable set of current state), the mapping is canceled and the corresponding memory of M' is marked as free. Conversely, when an inactive memory becomes active (by entering a state whose `avs` contains it), it is mapped to one of the free memories of M'. Some edge cases must be handled in practice, Section 5.1.4 describes the algorithm as implemented in the library created as part of this thesis.

Decreasing the number of MFA memories does not only result in decreased memory usage, but more importantly it leads into the set of possible configurations being significantly smaller and as a result the BFS search algorithm described in Section 4.2.1 will have improved time complexity. Using the BFS search, an input w can be matched against a regex α in time of the form $|\alpha||w|^{O(\texttt{avd}(\alpha))}$, which is exponential only in active variable degree [2]. The `avd` property thus induces a hierarchy of complexity classes and when it is bounded by a constant, the resulting regex subclass has polynomial matching problem.

4.4.2 Strong active variable degree

An extension of the active variable degree property was presented in [2]. As they point out, the disadvantage of `avd` property is that the $\triangleright_{\text{def}}$ relation used to compute it only tells which variables can be defined in a state, but does not differentiate whether the variables can be defined simultaneously. For example, regex $\alpha = (x_1\{\texttt{a}^+\} \vee x_2\{\texttt{b}\} \vee x_3\{\texttt{c}^*\})x_1x_2x_3$ has $\texttt{avd}(\alpha) = 3$, but only one variable can be defined at the same time. Another instance of such behavior was presented in Example 4.2. Consequently, the article introduces another property called *strong active variable degree*, which is able to recognize these cases.

To this end, the relation $\triangleright_{\text{def}}$ is redefined to be a relation over $\mathcal{P}(X) \times Q$, where X are variables and Q automaton states as in Definition 4.2, in the following way: $\{y_1, y_2, \ldots, y_l\} \triangleright_{\text{def}} q \Leftrightarrow \mathcal{R}(\alpha)$ can reach q by reading string w with $|w|_{o(y_i)} \geq 1$, for every $i \in [l]$. A strong active variable degree $\texttt{savd}(\alpha)$ can then be defined using this extended $\triangleright_{\text{def}}$ relation in an analogous way to how the `avd` was in Definition 4.2, see [2] for details.

However, practical relevance of the strong active variable degree property seems to be low, since it can not be computed efficiently (unless P=NP). Specially, deciding for a given $\alpha \in \text{RX}_{\Sigma,X}$ and $k \in \mathbb{N}$ whether $\texttt{savd}(\alpha) \leq k$ is coNP-hard [2].

4.4.3 Relation to variable distance

The active variable degree (`avd`) property shares some similarity with another complexity parameter called variable distance (`vd`) introduced in [37], which was already discussed in Section 3.4.2.

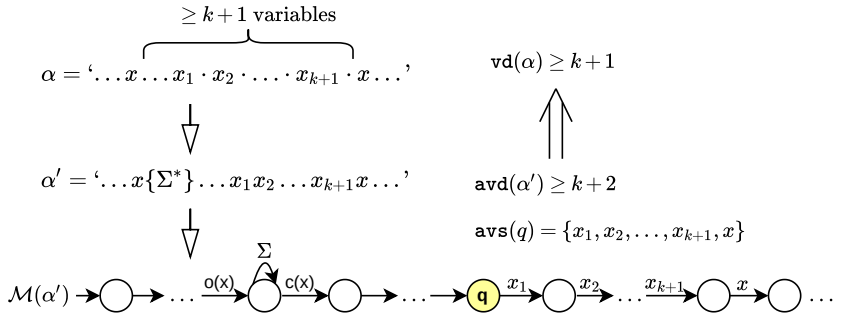


Figure 4.5: Relation between active variable degree and variable distance

The variable distance property is defined for terminal-free patterns (see Section 3.4.2) instead of regex and the article [37] also used a different automaton model than MFA. However, they point out that having multiple variable occurrences in the context of pattern languages corresponds to back-references in regex and thus captures the cause of computational complexity in regex matching [37]. Given a terminal-free pattern it is trivial to produce regex pattern (and thus also memory automaton) that describes the same language. Every first occurrence of a variable is replaced with variable binding in regex and any subsequent occurrences with variable recalls. As an example, for the terminal-free pattern $x_1 \cdot x_2 \cdot x_2 \cdot x_1 \cdot x_1$ the equivalent regex would be: $x_1\{\Sigma^*\}x_2\{\Sigma^*\}x_2x_1x_1$, where Σ stands for any symbol of the alphabet, in proper syntax of regex pattern over alphabet $\Sigma = \{a_1, a_2, \ldots, a_k\}$, the subexpression '$\Sigma^*$' would be written as '$(a_1 \vee a_2 \vee \ldots \vee a_k)^*$'. Both patterns describe language $L = \{uw^2u^2 \mid u, w \in \Sigma^*\}$.

The variable distance property could also be extended for regex by defining it to be the maximal number of occurrences of different variables between any two consecutive occurrences of a variable $x \in X$. Note that in context of regex, the term occurrence refers to both variable bindings and variable recalls. Using this extension, regex pattern produced for a terminal-free pattern in the previously described way would maintain the value of variable distance property.

The relation between values of these two properties will now be explored. If a terminal-free pattern has variable distance bounded by a constant, then the equivalent regex pattern also has constant active variable degree. Or specifically, it holds that $\texttt{vd}(\alpha) \leq k$ implies $\texttt{avd}(\alpha') \leq k + 1$, where $k \in \mathbb{N}$, α is a terminal-free pattern and α' an equivalent regex (i.e. $\mathcal{L}(\alpha) = \mathcal{L}(\alpha')$) constructed for α as described before. This can be proven using contradiction, the basic idea of the proof is illustrated on Figure 4.5. Suppose that $\texttt{vd}(\alpha) \leq k$, but $\texttt{avd}(\alpha') > k + 1$. Because $k \in \mathbb{N}$, it holds that $\texttt{avd}(\alpha') \geq k + 2$ and this means that in the MFA for α', there exists a state q with active variable

set containing at least $k + 2$ variables. To be in $\texttt{avs}(q)$, recall of each of these variables must be reachable from state q and this recall corresponds to variable occurrence in the original terminal-free pattern α. Let $x \in \texttt{avs}(q)$ be the variable whose occurrence corresponding to its first recall from q occurs last in α. Additionally, there must be some occurrence of x in α preceding all the variable occurrences corresponding to recalls. This is because $x \triangleright_{\text{def}} q$ and there was an occurrence of x in α corresponding to the variable binding that caused $x \triangleright_{\text{def}} q$. Because of how x was chosen, between these two consecutive occurrences of x in α are all the occurrences (corresponding to recalls) of the other $k+1$ variables from $\texttt{avs}(q)$. This by definition means that $\texttt{vd}(\alpha) \geq k+1$, which is a contradiction.

However, this relation does not hold in the opposite direction, even if `avd` is constant, the `vd` property may not be bounded by a constant. Suppose terminal-free pattern $\alpha = x \cdot x_1 \cdot x_1 \ldots \cdot x_n \cdot x_n \cdot x$ and the equivalent regex $\alpha' = x\{\Sigma^*\}(x_1\{\Sigma^*\}x_1)\ldots(x_n\{\Sigma^*\}x_n)x$, where $X = \{x_1, \ldots, x_n, x\}$ are variables and $n \in \mathbb{N}$. For each $i \in [n]$, variable x_i is recalled immediately after its binding in α' and thus any active variable set can contain at most variable x and one $x_j, j \in [n]$. Therefore $\texttt{avd}(\alpha') = 2$. However, because there are n different variables in α between the two occurrences of x, it holds that $\texttt{vd}(\alpha) = n$ and the `vd` property is not bounded by a constant.

Because of the above, the set of regex patterns with bounded active variable degree is a proper superset of patterns[23] with bounded variable distance. This also illustrates why the `avd` property is more useful for practical regex matching and further justifies the focus of this thesis on memory automata and the newer research e.g. in [2, 3].

4.5 Deterministic regex

Another subclass of regex with efficiently solvable matching problem is introduced and investigated in [3]. This subclass called *deterministic regex* is based on the notion of determinism in regular expressions with backreferences.

For the classical regular expressions, determinism is an established concept, see for instance [63], where such patterns are called *1-unambiguous*. Another common term, which is also used in [3], is *deterministic regular expressions*. Deterministic regular expressions can be defined using the already mentioned Glushkov's automaton construction [24]. If the Glushkov automaton for given regular expression is deterministic (i.e. it is a DFA), then the regular expression is also deterministic. Note that [63] established the concept using different definition, but they prove that it is equivalent to the definition using Glushkov automaton.

[23] As already discussed, the variable distance can also be extended for regex patterns.

Not only is the set of deterministic regular expressions a proper subset of (classical) regular expressions, but such patterns also have significantly reduced expressive power. In other words, the class of deterministic regular languages (i.e. languages that can be described by deterministic regular expressions) is a proper subclass of regular languages. This is shown in [63], where an example of language that can not be described by deterministic regular expression is provided. The pattern $\alpha = (\mathtt{0} \vee \mathtt{1})^*\mathtt{0}(\mathtt{0} \vee \mathtt{1})$ over $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ is not deterministic, and furthermore, the language $\mathcal{L}(\alpha) = \{u\mathtt{0}x \mid u \in \{\mathtt{0}, \mathtt{1}\}^*, x \in \{0, 1\}\}$ is not a deterministic regular language. Another example can be found in [3], where it is shown that the language $\mathcal{L}((\mathtt{ab})^*(\mathtt{a} \vee \varepsilon)) = \{(ab)^{\frac{1}{2}i} \mid i \geq 0\}$ also can not be described by any deterministic regular expression.

The deterministic regex defined in [3] are a proper extension of deterministic regular expressions. Deterministic regex patterns are proper subset of regex. They are defined using a variant of memory automata in a similar way to how determinism was formalized for classical regular expression using the Glushkov automaton. If memory automaton constructed from a given regex as described in [3] is deterministic, then the pattern is also deterministic regex. Determinism for memory automata was first established in [1], but the article [3] uses a modification of memory automata called *memory automata with trap state (TMFA)*.

TMFA are defined very similarly to MFA (see Definition 4.1), but they additionally have a special state called the *trap state*. This state is entered if a memory recall failure occurs during computation, i.e. when the contents of memory being recalled does not match the current input. When the trap state is entered, it is not left until the end of computation. TMFA where the trap state is not an accepting (final) state behaves in the same way as the corresponding standard MFA. However, setting the trap state to be final allows to accept complements of some regex languages. This leads to the property of *deterministic* TMFA languages being closed under complement[24], which is major reason for this modification. However, for the purpose of regex matching this difference from MFA is not as important. Intuitively, the deterministic TMFA are defined by restricting the transition function so that no more than one transition is possible from a configuration. [3]

The class of languages characterized by deterministic regex is a proper subset of regex languages, it is a proper superset of deterministic regular languages, and finally it is incomparable with regular languages [3]. This means that the practical usability of deterministic regex is limited, because there are languages that can not be matched by such patterns. The limitations of deterministic regular expressions were discussed before, the expressive power of deterministic regex is illustrated on the following examples.

[24]Note that regex languages do not have this property.

Example 4.3. The following patterns and languages are due to [3], note that the article uses a different syntax for regex: variable bindings are written as '$\langle x\colon \alpha\rangle$', variable recalls using '$\&x$'. The regex syntax from [2] will be used here.

Regex patterns $\alpha_1 = (x\{\mathtt{a}\} \vee \mathtt{a})$, $\alpha_2 = (\mathtt{a} \vee x)$ and $\alpha_3 = (x\{\varepsilon\} \vee \varepsilon)$ are not deterministic. On the other hand, $\alpha_4 = x\{(\mathtt{a} \vee \mathtt{b})^*\}\mathtt{c}x$ is a deterministic regex and describes language $\mathcal{L}(\alpha_4) = \{wcw \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$.

The regex $\alpha_5 = \mathtt{1}^+x\{\mathtt{0}^*\}(\mathtt{1}^+x)^*\mathtt{1}^+$ is not deterministic, but its language $\mathcal{L}(\alpha_5)$ can be described by deterministic regex $\alpha_6 = \mathtt{1}(\mathtt{1}^+ \vee (\mathtt{0}x\{\mathtt{0}^*\}\mathtt{1}^+(\mathtt{0}x\mathtt{1}^+)^*))$.

The formal language $L_1 = \{\mathtt{a}^{4i+1} \mid i \geq 0\} \cup \{\mathtt{a}^{4i} \mid i \geq 1\}$ can not be described by any deterministic regex. Similarly, the already mentioned language $L_2 = \{(ab)^{\frac{1}{2}i} \mid i \geq 0\}$ is not a deterministic regex language either. ♦

It is shown in [3] that the regex determinism can be checked in $O(|\Sigma||\alpha|k)$ for regex α with k variables and over alphabet Σ, and it is possible to construct the accepting TMFA at the same time. Furthermore, it is also shown that the matching problem for deterministic regex can be solved in $O(|\Sigma||\alpha|n+k|w|)$ for an input $w \in \Sigma^*$ with n occurrences of terminal symbols or variable references. An implementation of deterministic regex matching in Java programming language is available at [64].

4.6 Memory determinism

The deterministic regex explored in previous section are rather restricted, since they can not even describe all regular languages. Another form of memory automaton determinism is explored in [2]. Because memories are what causes the matching complexity of regex, the basic idea is to allow the nondeterminism for active states (as in classical NFA) and focus on restricting nondeterminism for memories. They introduce a subclass of regex called *memory deterministic regex*, which forces synchronization of memories between different computational branches of an automaton.

It is noted in [2] that formalizing such restriction does not seem to be possible using only local syntactic properties. Their definition of memory determinism uses properties of possible computations of MFA and is rather complicated. Intuitively, MFA is *memory deterministic* if all different computational branches (i.e. different active states in a computation step) do not differ in memory contents and statuses. Therefore the "harmless" nondeterminism of being in multiple active states is allowed, but the memories must have the same content and status – they must be *synchronized*. A regex α is *memory deterministic* if its memory automaton $\mathcal{M}(\alpha)$ is memory deterministic. See [2] for precise definitions and proofs of memory deterministic regex properties.

Unlike deterministic regex, memory deterministic regex properly extend the classical regular expressions – any regular expression is memory deterministic (since it has no memories). The subclass of regex languages that can be described by memory deterministic regex is a proper superset of regular languages.

Deciding whether a given regex pattern α is memory deterministic can be done in time $O(|\alpha|^5)$ as shown in [2]. Additionally, the matching problem for memory deterministic regex can be solved efficiently, [2] presented an algorithm that for given memory deterministic regex α over Σ with k variables and input $w \in \Sigma$, decides $w \in \mathcal{L}(\alpha)$ in time $O(|w||\alpha|^3(|\Sigma| + k))$. The algorithm resembles the online NFA simulation discussed earlier. The active configurations are kept by storing all the active states, but the memory contents and statuses are shared for all active computational branches. This can be done because memory determinism property ensures that different branches of computation do not differ in memory contents or statuses.

An elaborate example of memory-deterministic regex is provided in the appendix of [2]. A simple example of pattern that is memory-deterministic is $\alpha = x\{\mathtt{a}^+\}\mathtt{b}x$. By contrast, pattern $\beta = \mathtt{a}^*x\{\mathtt{a}^+\}\mathtt{a}x$ is not memory-deterministic, because there are non-deterministic choices on when the memory is opened and closed, resulting in possible difference in memory content and status between the computational branches.

As was the case with active variable degree, the memory determinism property can also be extended to describe larger subset of regex. An improvement called *synchronized* memory automaton property is investigated in the article. The matching complexity of the regex subclass induced by this property is the same as in the case of memory deterministic regex. Unfortunately, deciding whether this improved property holds for a given MFA is coNP-hard. [2]

Chapter 5

Implementation

This chapter describes the regular expression engine (library) based on memory automata that was implemented as part of this thesis. First, a high level description of how the memory automaton model was implemented is provided. The implemented matching algorithms are formalized, and an alternative algorithm to compute active variables is also discussed. Additionally, our extension to memory automaton model needed for matching counting constraints, a feature often found in practical regular expressions, is described. Then, the architecture of the solution is discussed and key implementation details are also provided. Finally, features and regular expression extensions supported by the engine are described briefly.

5.1 Memory automata for practical regular expressions

The memory automaton model described in Chapter 4 was used as a basis for the implementation. A k-memory automaton $M = (Q, \Sigma, \delta, q_0, F)$ can be represented as an edge-labeled directed graph, with vertex for each state $q \in Q$, and each x-transition $q \xrightarrow{x}_\delta p$ represented as an edge from q to p labeled with x.

To construct memory automaton for a given regex pattern, the algorithm discussed in Section 4.2 is used. The implementation uses LR parsing [62] to parse the given pattern into its syntax tree, and then the memory automaton is constructed. The construction is done recursively, by traversing the syntax tree in post-order, starting in the root node. In each recursive call, the automaton parts for the at most two child nodes of the current node are constructed, then these parts are combined as illustrated in Figure 4.2. This way, automaton for the whole pattern is incrementally constructed. Additional details about the grammar and the parser implementation are given later.

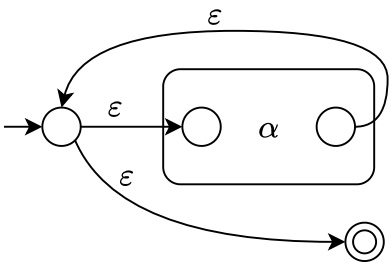


Figure 5.1: MFA construction for pattern of the form α^*

The construction rules as defined in [2] are generally followed. An exception is the concatenation construct, where the redundant t_m node (the node between automata parts for α and β in Figure 4.2e) is omitted and an ε-transition is added directly between the two parts. This maintains properties of the construction discussed below.

Additionally, patterns of the form α^* are not constructed by treating them as $\alpha^+ \vee \varepsilon$, but an equivalent simplified construction depicted on Figure 5.1 is used instead. This still maintains the required properties and results in lower number of states in the constructed automaton.

As discussed in Section 4.2, memory automata constructed for regex patterns via the algorithm have at most two outgoing transitions from each state, both labeled by the same symbol. Because of this, the automaton can be stored by keeping for each vertex (state) the symbol, for which outgoing edges exists, and the at most two vertices where the edges lead. Therefore, as [2] also points out, such memory automata satisfy the following properties:

1. There are $O(|Q|)$ edges in the automaton graph representation, which also implies $|\Sigma| = O(|Q|)$. Therefore, the automaton can be stored using $O(|Q|)$ space.
2. For a given $q \in Q$ and $x \in \Sigma$, all (at most two) states $p \in Q$, such that $p \in \delta(q, x)$, can be retrieved in constant time.
3. A BFS traversal of the states can be performed in $O(|Q|)$ time.

As mentioned in Section 4.2, there is a constant number of states constructed for each syntax tree node, therefore it also holds that $|Q| = O(|\alpha|)$, for regex pattern α. The resulting automaton will also have one initial and one final state, both will be kept.

5.1.1 Matching algorithms

Having constructed the MFA for given regex pattern, the algorithm described in Section 4.2.1 can be used to match an input string (i.e. given for input w,

decide $w \in \mathcal{L}(\alpha)$). The algorithm is based on a breadth-first search (BFS) on the set of automaton configurations. The search starts in the initial configuration and traverses the set of configurations until it reaches an accepting one.

The BFS matching algorithm was implemented as a typical breath-first search. The searched space of configurations is not constructed explicitly, but instead the "neighboring" configurations are computed dynamically during search. The neighbor function that given a configuration returns all the configurations reachable from it in one step is an important part of the algorithm. This function realizes the transition relation $\vdash_M$ from Definition 4.1, the implementation follows this definition closely.

Another alternative algorithm based on backtracking was also implemented. The algorithm is similar to the one described in Section 3.3. The implementation also employs the technique described in Section 3.3.1 to prevent infinite loops by caching some configurations. Specifically, when there are multiple ε-transitions from current state, the current configuration is cached. When cached configuration is visited again, we know its neighbors were already searched, so they are not searched again. Additionally, an option to cache all visited configurations is provided, in which case the algorithm behaves as a standard depth-first search (DFS) algorithm.

Both matching by BFS and DFS search guarantee the time complexity discussed in Section 4.2.1. The backtracking algorithm, which caches configurations only when necessary, behaves similarly as the DFS on memory automata constructed from regex, but it has significantly lower memory usage.

5.1.2 Configurations representation and implementing memories

According to Definition 4.1, memory automaton configurations are of the form $(q, w, (u_1, r_1), \ldots, (u_k, r_k))$. Remaining part of the input w and the memory contents u_i can be stored as pointers to the original input string. Memory statuses r_i are boolean variables. The current state can be kept as a reference to the corresponding node in the graph representation of MFA.

The memory contents are represented as two pointers into the input string, first pointing at the beginning of the stored substring and the second pointing at the end. It was already discussed that u_i always contains substring of the input. When an $o(x)$ transition is taken, the beginning pointer of corresponding memory x is set to current input position. The second pointer is also set to the same value, as the memory contents is initially empty.

As per definition of the $\vdash_M$ relation, when a memory is open (i.e. $r_x = \mathsf{0}$), all the matched (consumed) symbols are appended to its content. However,

the implementation does not do this explicitly. The ending pointer is instead set when a $c(x)$ transition is taken. It will now be explained that this still maintains correctness of the matching algorithms. Because of how the MFA is constructed for regex variable binding (see Figure 4.2b), each taken $o(x)$ transition in the resulting automaton is eventually followed by a $c(x)$ transition. Afterwards, content of memory x must be equal to the string consumed between taking these two transitions. Because the beginning pointer is set during $o(x)$ transition and the end pointer during $c(x)$, the stored string will be correct after taking the $c(x)$ transition, as the pointers enclose exactly the consumed string.

However, before the $c(x)$ transition is taken, the content behaves as ε, while according to definition it should instead accumulate consumed symbols. It remains to see that this does not cause any problems for the matching algorithms. Firstly, there can not be any memory recall transition before $c(x)$ is taken, since Definition 2.1 prohibits any subpatterns of the form $x\{\dots x\dots\}$. Secondly, it will not happen that a non-visited configuration would be wrongly considered visited, because if two configurations differed in the content of memory x after taking $o(x)$, one of them must have consumed different part of input, and therefore they must also differ in the remaining input (which is also part of configuration).

Thanks to this, we do not have to iterate over all memories and potentially update their memory contents during each transition, but it is only done when either $o(x)$ or $c(x)$ transition is taken.

5.1.3 Computing active variable sets

As discussed in Section 4.4, to efficiently match regex with bounded active variable degree, the set of active variables needs to be computed for each state. This can be done by first computing the $\triangleright_{\text{def}}$ and $\triangleright_{\text{call}}$ relations. An algorithm was described in [2], however, as explained in Section 4.4, following this method exactly seems impractical for implementing regex matching. Alternative algorithm used in our implementation, which is based on BFS traversal of the MFA graph representation is presented here.

The active variable sets are computed by first finding all variables $x \in X$ that are in $\triangleright_{\text{def}}$ relation with this state, and then doing the same for the $\triangleright_{\text{call}}$ relation. Variable is then active in a state if it is contained in both these sets. This works, because from definition $\texttt{avs}(q) = \{x \mid x \triangleright_{\text{def}} q\} \cap \{x \mid q \triangleright_{\text{call}} x\}$. We compute `avs` by first computing these two sets for each state and then performing intersection between them. The sets will be denoted $X_{\triangleright_{\text{def}}}(q)$ and $X_{\triangleright_{\text{call}}}(q)$ respectively.

Algorithm 2 formalizes this method of computing `avs`. It is assumed that MFA constructed from a regex using the algorithm described earlier is given

Algorithm 2: Active variable sets computation

Input : MFA $M = (Q, \Sigma, \delta, q_0, F)$, represented as directed graph $G(M)$
Output: $\texttt{avs}(q)$ for each $q \in Q$

```
1  for q ∈ Q do X_{▷def}(q) ← ∅
2  for q ∈ Q do X_{▷call}(q) ← ∅
   /* first compute the ▷def relation                    */
3  foreach q ∈ Q do
4  |  if ∃x ∈ X, |δ(q, o(x))| ≥ 1 ∧ x ∉ X_{▷def}(q) then
5  |  |  run BFS on G(M) starting from q, for each visited p ∈ Q set:
   |  |    X_{▷def}(p) ← X_{▷def}(p) ∪ {x}
6  |  end
7  end
   /* compute the ▷call relation next                    */
8  Let G̃^R(M) be the reverse of G(M) with edges labeled o(x) removed
9  foreach q ∈ Q do
10 |  if ∃x ∈ X, |δ(q, x)| ≥ 1 ∧ x ∉ X_{▷call}(q) then
11 |  |  run BFS on G̃^R(M) starting from q, for each visited p ∈ Q set:
   |  |    X_{▷call}(p) ← X_{▷call}(p) ∪ {x}
12 |  end
13 end
   /* compute active variable sets as intersections      */
14 for q ∈ Q do avs(q) ← X_{▷def}(q) ∩ X_{▷call}(q)
```

as an input. It is also assumed that the automaton is represented as an edge-labeled directed graph, as described at the beginning of Section 5.1. This graph representation is denoted $G(M)$ in the pseudocode.

First the $X_{\triangleright_{\text{def}}}$ sets are computed. Following Definition 4.2, $x \triangleright_{\text{def}} q$ if the state q can be reached from initial state via a path containing at least one $o(x)$ transition. The loop on lines 3–7 finds all states with outgoing $o(x)$ transition and runs BFS search from each one, x is added to $X_{\triangleright_{\text{def}}}$ set of every reachable state. This correctly computes the $X_{\triangleright_{\text{def}}}$ sets, because if a state is reached using the BFS search, there exists a path to it containing[25] at least one edge labeled with $o(x)$, which corresponds to $o(x)$ transition in the MFA. Additionally, all states in a MFA constructed using the rules from [2] are reachable from the initial state. Therefore, any x added to $X_{\triangleright_{\text{def}}}(p)$ on line 5 satisfies $x \triangleright_{\text{def}} p$. The BFS is run for all $o(x)$ transitions, so every variable x is added to all the corresponding sets.

[25] As established before, all transitions from a state are of the same type in MFA for regex and thus all paths found by the BFS search have its first edge labeled $o(x)$.

The $x \notin X_{\triangleright_{\text{def}}}(q)$ condition on line 4 ensures that the search is not started from states already visited in a previous BFS run. If $x \in X_{\triangleright_{\text{def}}}(q)$, the state q was visited in previous BFS and so were by extension any other states reachable from it, hence x was already added to all their sets and it is not necessary to run the BFS from q.

Then the $X_{\triangleright_{\text{call}}}$ sets are computed in a similar way. It follows from Definition 4.2 that $q \triangleright_{\text{call}} x$ if there is a path from q that does not contain any edge labeled with $o(x)$ and its last edge is labeled with variable x. First the reverse of graph $G(M)$ is computed and all edges labeled with $o(x)$ are removed, for any variable x. The resulting graph is denoted $\widetilde{G}^{\text{R}}(M)$. Then, on lines 9–13, breadth-first search is run on $\widetilde{G}^{\text{R}}(M)$ from each state q that has an outgoing edge labeled with variable x (which corresponds to variable recall). All visited states have the variable x added to their $X_{\triangleright_{\text{call}}}$ sets.

This works similarly as the previous computation of $X_{\triangleright_{\text{def}}}$ sets. If a state is visited during the BFS run from line 11, there is a path in the original graph $G(M)$, which starts in q, ends with an x-labeled edge (for variable x), and does not contain any $o(x)$ edge, because such edges were removed from the searched graph $\widetilde{G}^{\text{R}}(M)$. The $x \notin X_{\triangleright_{\text{call}}}(q)$ condition on line 10 again prevents an already visited state from being a source of another BFS run, because all reachable states were already visited before in such case.

After both $X_{\triangleright_{\text{def}}}$ and $X_{\triangleright_{\text{call}}}$ sets are computed, intersection between the two corresponding sets is performed for each state, which according to definition correctly computes the active variable sets $\texttt{avs}(q)$. Active variable degree can then be taken as maximum from `avs` set sizes.

As discussed before, thanks to the properties of MFA for regex, one BFS traversal runs in time $O(|Q|)$, and it also holds that $|Q| = O(|\alpha|)$. Because of the conditions on lines 4 and 10, for each variable $x \in X$, every state $q \in Q$ is visited only once. There is also a constant number of edges from each state and we can check $|\delta(q, o(x))| \geq 1$ in $O(1)$ time. All the BFS runs thus would take time $O(|X| \cdot |\alpha|)$ without considering the operations on sets. The complexity of set operations depend on the chosen representation. Because the number of variables $|X|$ is known in advance, the $X_{\triangleright_{\text{def}}}$ and $X_{\triangleright_{\text{call}}}$ sets, as well as sets of active variables can be stored in an array of boolean variables (bit array) for each state. Such representation yields $O(1)$ complexity for both insertion (set bit to true) and look-up operations. Additionally, intersection can be computed in time $O(|X|)$. This representation was used in the implementation.

In conclusion, the computations on lines 3–7 and 9–13 take $O(|X| \cdot |\alpha|)$ time. The graph $\widetilde{G}^{\text{R}}(M)$ can be computed in $O(|\alpha|)$, because G(M) has $O(|\alpha|)$ vertices and constant number of edges from each, meaning that there are also $O(|\alpha|)$ edges in the graph. Finally, the computation of `avs` on line 14 is done in $O(|X| \cdot |\alpha|)$ time. Therefore, Algorithm 2 has $O(|X| \cdot |\alpha|)$ time complexity.

Both graphs $G(M)$ and $\widetilde{G}^{\mathrm{R}}(M)$ require $O(|\alpha|)$ memory. The $X_{\triangleright_{\mathrm{def}}}$, $X_{\triangleright_{\mathrm{call}}}$, and `avs` sets are stored using $|Q| \cdot O(|X|) = O(|\alpha| \cdot |X|)$ space. Keeping track of visited states during BFS run requires additional $O(|\alpha|)$ memory. The algorithm thus has memory complexity $O(|X| \cdot |\alpha|)$.

5.1.4 Matching patterns with bounded active variable degree

The implementation employs the algorithm from [2] discussed in Section 4.4.1, to efficiently match regex patterns with bounded active variable degree. The algorithm uses active variable sets computed earlier to simulate memory automaton using another automaton with (potentially) lower number of memories – only $\texttt{avd}(\alpha)$ memories are needed. A high level description of the simulation was presented in the article, the implemented algorithm follows this description closely. Because pseudocode was not given in the article, we include pseudocode of our implementation in Algorithm 3 for the sake of completeness. See also [2] for proof of correctness.

The implementation simulates computation of the MFA(k) M using only $\texttt{avd}(\alpha)$ memories by working with modified configurations, which include the memory list $\mathfrak{M}$ as described in [2]. The configurations will be of the form $(q, w, (u_1, r_1), \ldots, (u_{k'}, r_{k'}), \mathfrak{M})$, where $k' = \texttt{avd}(\alpha)$. In fact, they are configurations of a k'-memory automaton with added mapping $\mathfrak{M}$ between the k' "new" memories and the k "old" memories of the automaton M. The new memories handle subset of the old memories that are currently active. Algorithm 3 formalizes how the transitions are simulated, it effectively describes transition relation between the modified configurations.

Let MFA(k) M be an automaton constructed from regex pattern α, with $k' = \texttt{avd}(\alpha)$, that is simulated using k' memories. The modified configurations are of the form $c = (q, w, (u_1, r_1), \ldots, (u_{k'}, r_{k'}), \mathfrak{M})$, where $\mathfrak{M} \in ([k] \cup \bot)^{k'}$ is the memory list. In the pseudocode of Algorithm 3, the part of such modified configuration c without $\mathfrak{M}$ is denoted $\texttt{base}(c)$. The initial configuration on input w is $(q_0, w, (u_1, r_1), \ldots, (u_{k'}, r_{k'}), (\bot, \ldots, \bot))$.

5.1.5 Extending memory automata for counting constraints

An extension commonly found in practical regular expressions are counting constraints, which were already discussed in Section 2.6. They allow to match a subpattern repeatedly, with the number of repetitions fixed or from given allowed range. First, regex patterns extended with counting constraints are established. Then, our extension to memory automata for efficient matching of such patterns is described.

Because the curly brackets '{' and '}' used commonly in practice to denote counting constraints are already used in regex to mark the bounded subpattern in variable bindings, for our definition angle brackets '⟨' and '⟩' will be used

Algorithm 3: Simulating transitions of automaton with bounded active variable degree

Input : MFA(k) $M = (Q, \Sigma, \delta, q_0, F)$ for regex α, $\texttt{avd}(\alpha) = k'$, $\texttt{avs}(q)$ for each $q \in Q$, current configuration $c = (q, w, (u_1, r_1), \ldots, (u_{k'}, r_{k'}), \mathfrak{M})$, and the simulated transition $q \xrightarrow{x}_{\delta} p$

Output: resulting configuration $c' = (p, w', (r'_1, u'_1), \ldots, (r'_{k'}, u'_{k'}), \mathfrak{M}')$

if $x = o(l), l \in [k]$ **then** `/* memory open transition */`
  if $l \in \texttt{avs}(p)$ **then** `/* allocate new memory and open it */`
    $l' \leftarrow min\{i \mid \mathfrak{M}[i] = \bot\}$
    $\mathfrak{M}[l'] \leftarrow l$
    $x' \leftarrow o(l')$
  else `/* do not open unused memory (not in avs) */`
    $x' \leftarrow \varepsilon$
  end
else if $(x = c(l) \vee x = l), l \in [k]$ **then** `/* memory recall/close */`
  if $\exists i, \mathfrak{M}[i] = l$ **then**
    $l' \leftarrow \{i \mid \mathfrak{M}[i] = l\}$
  else
    $l' \leftarrow \bot$
  end
  if $l \in \texttt{avs}(q) \wedge l' \neq \bot$ **then**
    if $x = c(l)$ **then** $x' \leftarrow c(l')$ **else** $x' \leftarrow l'$
  else
    $x' \leftarrow \varepsilon$
  end
else `/* ordinary (non-memory) transition */`
  $x' \leftarrow x$
end
execute transition $q \xrightarrow{x'} p$ on configuration $\texttt{base}(c)$, save the result in $\hat{c}$
foreach $\{i \mid \mathfrak{M}[i] \notin \texttt{avs(p)}\}$ **do** `/* free non-active memories */`
  $\mathfrak{M}[i] \leftarrow \bot$
end
return *configuration* $\hat{c}$ *with* $\mathfrak{M}$

instead. However, the implementation accepts commonly used PCRE-like syntax, which uses curly brackets for these constructs.

Definition 5.1 (regex with counting constraints)**.** The syntax of *regex with counting constraints* over Σ and X (denoted $\mathrm{RXC}_{\Sigma,X}$) is based on the rules from Definition 2.1. The recursive rules 1–4 from that definition are the same for this definition of $\mathrm{RXC}_{\Sigma,X}$, with three additional rules:

5. For every $\alpha \in \mathrm{RXC}_{\Sigma,X}$ and $a \in \mathbb{N}_0$, $(\alpha)\langle a\rangle \in \mathrm{RXC}_{\Sigma,X}$ and $var((\alpha)\langle a\rangle) = var(\alpha)$.

6. For every $\alpha \in \mathrm{RXC}_{\Sigma,X}$ and $a, b \in \mathbb{N}_0, a < b$, $(\alpha)\langle a,b\rangle \in \mathrm{RXC}_{\Sigma,X}$ and $var((\alpha)\langle a,b\rangle) = var(\alpha)$.

7. For every $\alpha \in \mathrm{RXC}_{\Sigma,X}$ and $a \in \mathbb{N}_0$, $(\alpha)\langle a,\rangle \in \mathrm{RXC}_{\Sigma,X}$ and $var((\alpha)\langle a,\rangle) = var(\alpha)$.

The semantics are defined in the following way. If pattern $\alpha \in \mathrm{RXC}_{\Sigma,X}$ does not contain '$\langle$' or '$\rangle$' (i.e. no subpattern was produced by rule 5, 6 or 7), then also $\alpha \in \mathrm{RX}_{\Sigma,X}$ and the language $\mathcal{L}(\alpha)$ described by α is defined using the Definition 2.2 of regex semantics. Otherwise, if pattern α was produced by rule 5, then $\alpha = (\beta)\langle a\rangle$ and the language described by α is defined to be $\mathcal{L}((\beta)\langle a\rangle) = \mathcal{L}(\beta^a)$, i.e. the language of pattern β repeated a times. Additionally, if $\alpha = (\beta)\langle a,b\rangle$ was produced by rule 6, its language is defined as $\mathcal{L}((\beta)\langle a,b\rangle) = \mathcal{L}((\beta)\langle a\rangle \vee (\beta)\langle a+1\rangle \vee \ldots \vee (\beta)\langle b\rangle)$. Finally, if $\alpha = (\beta)\langle a,\rangle$ was produced by rule 7, its language is $\mathcal{L}((\beta)\langle a,\rangle) = \mathcal{L}((\beta)\langle a\rangle \cdot \beta^*)$. Applying these three rules recursively eventually leads to pattern without '$\langle$' or '$\rangle$' and the regex semantics can be used. △

As seen from Definition 5.1, adding counting constraint does not change semantic power of regex, meaning that this addition is a mere "syntactic sugar". Since such patterns can, by definition, be transformed into regex, counting constraints could be supported simply by using the above rules and building an equivalent regex, then matching as usual. However, the problem with this approach is that the resulting regex would potentially be very large.

Example 5.1. Pattern $\alpha_1 = (\mathtt{ab}^*)\langle 3\rangle \in \mathrm{RXC}_{\{\mathtt{a},\mathtt{b}\},\emptyset}$ describes language of regex pattern $\alpha_1' = \mathtt{ab}^*\mathtt{ab}^*\mathtt{ab}^*$, which is $L_1 = \{\mathtt{ab}^i\mathtt{ab}^j\mathtt{ab}^k \mid i,j,k \in \mathbb{N}_0\}$.

Another pattern $\alpha_2 = (x_1\{\mathtt{a}^+\}\mathtt{b}x_1)\langle 1,2\rangle \in \mathrm{RXC}_{\{\mathtt{a},\mathtt{b}\},\{x_1\}}$ has an equivalent regex pattern $\alpha_2' = (x_1\{\mathtt{a}^+\}\mathtt{b}x_1) \vee (x_1\{\mathtt{a}^+\}\mathtt{b}x_1x_1\{\mathtt{a}^+\}\mathtt{b}x_1)$, the described language is $L_2 = \{\mathtt{a}^i\mathtt{ba}^i \mid i \in \mathbb{N}\} \cup \{\mathtt{a}^j\mathtt{ba}^{j+k}\mathtt{ba}^k \mid j,k \in \mathbb{N}\}$. ♦

For pattern of the form $\alpha = (\beta)\langle k\rangle \in \mathrm{RXC}_{\Sigma,X}$, the equivalent regex would be $\alpha' = \beta^k$. The size of this regex is then $|\alpha'| = k \cdot |\beta|$, and if the pattern was

matched using memory automaton, the resulting MFA would grow proportionally to the value of k, which would be rather problematic for large k. In fact, if the number of repetitions in pattern α is given using base-b positional numeral system, the size of α' would be $O(b^{|\alpha|})$ and therefore exponential in the size of the original pattern α. Furthermore, this can be problematic in practice even for smaller number of repetitions if the subpattern is large and even more so if a range $\langle a, b\rangle$ is given.

To match regex with counting constraints efficiently, we propose extending memory automata with *counters*. The basic idea is that patterns such as $(\beta)\langle k\rangle$ can be matched using a similar construction as for β^+, but counting how many times the "inner" subpattern β got matched and allowing to take the transition leading to the final state only when this count is equal to k. Similarly, patterns of the form $\langle a, b\rangle$ can be matched in a similar way, but checking the count against an interval instead.

Intuitively, this corresponds to replacing the k "copies" of sub-automaton for β, which would be constructed if $(\beta)\langle k\rangle$ was treated as β^k, with a single instance of sub-automaton for β and keeping a counter that is incremented each time the subpattern is matched. This is somewhat similar to how counting constraints are handled in [35], although it is unclear whether repetitions of more complicated subpatterns (as opposed to repetitions of single characters and character classes) were considered in that paper. Also a different matching algorithm based on online NFA simulation was used in the article, see Section 3.4.1.

We propose adding integer variables called *counters* to memory automaton configurations. Every counting constraint is then associated with a counter. Each time the repeated subpattern[26] is matched, the corresponding counter is incremented. Transition outside the part of automaton implementing the counting constraint will then be taken only if value of the associated counter is either equal to k for constraint $(\beta)\langle k\rangle$, inside $[a, b]$ interval for $(\beta)\langle a, b\rangle$, or is greater than a for $(\beta)\langle a,\rangle$.

Multiple counters may be required to match patterns such as $((\mathtt{a})\langle k\rangle\mathtt{b}^+)\langle l\rangle$. When matching the subpattern $(\mathtt{a})\langle k\rangle$ in this pattern, a counter is required to keep the number of matched `a` symbols, but another counter is also needed for matching the whole subpattern inside the enclosing $(\ldots)\langle l\rangle$ constraint.

On the other hand, once a subpattern of the form $(\beta)\langle k\rangle$ (or other variants of counting constraint) gets matched, value of the corresponding counter is no longer required for matching rest of the pattern. Therefore, it is possible to reuse counters for matching multiple counting constraints provided that they are not nested as they were in the pattern from previous paragraph. For example, to match $(\mathtt{a})\langle k\rangle(\mathtt{bc}^+)\langle l\rangle$, only one counter is sufficient. It would first

[26] e.g. β in $(\beta)\langle k\rangle$

store the number of matched `a` symbols (until its value reaches k), and then it would be reused for matching $(\texttt{bc}^+)\langle l \rangle$. This sharing of a counter can be done as long as the two counting constraints are not nested (and thus are independent), i.e. if one is not parent of the other in the pattern's syntax tree.

Now the counters in memory automata will be established formally, by introducing an extension of the MFA model from Definition 4.1. Configurations are extended with counters, and two special symbols that represent two new types of transitions are added.

Definition 5.2 (memory automaton with counters)**.** For $l \in \mathbb{N}$, we define $\Xi_l = \{\mathrm{INC}(x, a, b), \mathrm{OUT}(x, a, b) \mid x \in [l], a \in \mathbb{N}_0, b \in \mathbb{N}_0 \cup \{\infty\}, a \leq b\}$, where ∞ is a special value satisfying $\forall x \in \mathbb{N}_0, x < \infty$. Additionally, we will use $\Sigma_{\varepsilon,k}$ and Γ_k as established in Definition 4.1.

For $k, l \in \mathbb{N}$, a *k-memory automaton with l counters* (denoted $\mathrm{CMFA}(k, l)$) is syntactically an NFA $(Q, \Delta', \delta, q_0, F)$, where $\Delta' = \Sigma_{\varepsilon,k} \cup \Gamma_k \cup \Xi_l$. The semantics are defined by extending semantics of k-memory automaton.

Configurations are of the form $c = (q, w, (u_1, r_1), \ldots, (u_k, r_k), (C_1, \ldots, C_l))$, where $\texttt{base}(c) = (q, w, (u_1, r_1), \ldots, (u_k, r_k))$ is a configuration of $\mathrm{MFA}(k)$ on the same alphabet Σ, and $C_i \in \mathbb{N}_0$ for $i \in [l]$ are values of the counters. Initial configuration on input $w \in \Sigma^*$ is $(q_0, w, (\varepsilon, \texttt{C}), \ldots, (\varepsilon, \texttt{C}), (0, \ldots, 0))$. Configuration c is accepting if $q \in F$ and $w = \varepsilon$.

The transition relation $\vdash_M$ is defined on the set of configurations as follows. For $\mathrm{CMFA}(k,\ l)$ configurations $c = (q, w, (u_1, r_1), \ldots, (u_k, r_k), (C_1, \ldots, C_l))$ and $c' = (p, w', (u'_1, r'_1), \ldots, (u'_k, r'_k), (C'_1, \ldots, C'_l))$, it holds $c \vdash_M c'$ if one of these conditions apply:

1. $\texttt{base}(c) \vdash \texttt{base}(c')$ as per Definition 4.1, and $\forall j \in [l], C'_j = C_j$.
2. $p \in \delta(q, \mathrm{INC}(x, a, b)) \wedge w' = w \wedge (\forall i \in [k], (u'_i, r'_i) = (u_i, r_i)) \wedge$
   $\wedge\, (\forall j \neq x, C'_j = C_j)$, and either of these apply:
   a) $b \in \mathbb{N}_0 \wedge C_x < b$, then $C'_x = C_x + 1$,
   b) $b = \infty \wedge C_x < a$, then $C'_x = C_x + 1$,
   c) $b = \infty \wedge C_x = a$, then $C'_x = C_x$.
3. $p \in \delta(q, \mathrm{OUT}(x, a, b)) \wedge w' = w \wedge (\forall i \in [k], (u'_i, r'_i) = (u_i, r_i)) \wedge$
   $\wedge\, (\forall j \neq x, C'_j = C_j) \wedge (a \leq C_x \leq b)$, then $C'_x = 0$.

The accepted language, computations of CMFA, and other concepts are defined in an analogous way to how they were defined for MFA. △

To construct CMFA accepting language of a given regex with counting constraints, an extension of the rules from Section 4.2 will be used. Figure 5.2

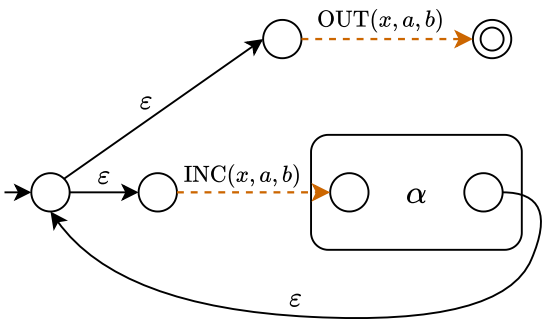


Figure 5.2: Construction of CMFA for cregex pattern of the form $(\alpha)\langle a, b\rangle$. If the given pattern is of the form $(\alpha)\langle a\rangle$ instead, the same construction will be used with $b = a$. Similarly, for constraint $(\alpha)\langle a,\rangle$, we set $b = \infty$.

shows how an automaton constructed for each of the counting constraint variants will look like. Structure of the resulting automaton will satisfy the properties discussed at the beginning of Section 5.1, because there are still at most two outgoing transition from each state and both are in that case labeled with the same symbol. The ε-transitions in the diagram are there specifically for the purpose of satisfying these properties. Adding this rule thus preserves the complexity characteristics of memory automata outlined earlier. Using the rule along with the other construction rules described in Section 4.2, any cregex pattern can be recursively transformed into an equivalent CMFA.

Two special transitions can be seen on the diagram in Figure 5.2. The transition labeled with `INC`(x, a, b) leads into the part of automaton constructed from subpattern α that is subjected to the counting constraint. According to rule `2` in Definition 5.2, this transition can be taken only if value of the corresponding counter C_x is lower than b, where b is in fact the upper bound on number of allowed subpattern α repetitions. When the transition is taken, the corresponding counter is incremented. This is natural, since the purpose of the counter is to keep track of how many times the subpattern was matched and after taking this transition the subpattern will be matched before returning back to the source state. If the constraint is of the form $(\alpha)\langle a,\rangle$, it is only incremented until it reaches the value of a, after that `INC` transition does not increase the value further. This is done to restrict the number of possible configurations, which leads to better complexity properties, since time complexity of the matching algorithms described earlier depends on the size of the set of possible configurations.

The second special transition labeled with `OUT`(x, a, b) can be taken only if the counter's value is inside allowed range (e.g. $[a, b]$ for $(\alpha)\langle a, b\rangle$) defined by the constraint, as described by rule `3` in Definition 5.2. After this transition

is taken, the corresponding counter is set to zero so that it can be potentially reused for another counting constraints appearing further in the pattern.

The construction from Figure 5.2 has the same effect as repeating the pattern (and the corresponding automaton part) explicitly. Instead of explicitly copying the part for subpattern α, we reuse it and keep count of how many times the subpattern was matched, effectively matching the same pattern. When handling counting constraint of type $(\alpha)\langle a,\rangle$, the counter stops incrementing at a and then the automaton part behaves as α^* construct, which is consistent with the fact that $(\alpha)\langle a,\rangle$ can be rewritten to $(\alpha)\langle a\rangle\alpha^*$, according to Definition 5.1.

The only issue left is how to choose the counter x in the automaton construction from Figure 5.2. As discussed earlier, nested counting constraints can not be handled by the same counter. Otherwise, we can reuse one counter for multiple counting constraints if they are independent. Analogously as for memories, it is desirable to minimize number of counters used by automaton. This is because, as per Definition 5.2, counters are part of configurations and so they contribute to the configuration size and thus the memory needed to store the configurations. More importantly, having more counters also increases size of the searched space of automaton configurations.

The implementation chooses counters by assigning counter number 1 to the most nested counting constraint subexpression (i.e. whose node in the syntax tree has the largest distance from root) and any enclosing counting constraint will have counter number larger by one. To define this rule precisely, we set $\texttt{next\_counter}(\gamma) = 1$ for any cregex pattern γ produced by rule `1` or rule `3` from Definition 2.1. If γ was produced by rule `2`, we set $\texttt{next\_counter}(\gamma) = max\{\texttt{next\_counter}(\alpha), \texttt{next\_counter}(\beta)\}$. For rule `4` we set $\texttt{next\_counter}(\gamma) = \texttt{next\_counter}(\alpha)$. Finally, if γ was produced by rule `5`, `6` or `7` from Definition 5.1, we choose the counter as $x = \texttt{next\_counter}(\alpha)$ and set $\texttt{next\_counter}(\gamma) = \texttt{next\_counter}(\alpha) + 1$. It can be seen that the CMFA constructed from pattern α as described will have $\texttt{next\_counter}(\alpha) - 1$ counters.

As described earlier in Section 5.1, the automaton construction is done recursively, by first constructing the automata parts (fragments) for child nodes and then combining them using construction rule according to the type of current node. Implementation of the rules for assigning counters as described above is trivial, we can keep the values of `next_counter` in the automaton fragments and compute the value according to child nodes and node type. Algorithm 4 formalizes how the assignment of counters is done and also illustrates the overall automaton construction process. We use the same symbols for denoting syntax tree node types as were used in [2], and additionally use $[\langle a\rangle]$, $[\langle a, b\rangle]$, and $[\langle a,\rangle]$ to denote the tree types of counting constraints.

Algorithm 4: Construction of CMFA and assignment of counters

Input : Pattern $\alpha \in \mathrm{RXC}_{\Sigma,X}$, its syntax tree $\mathcal{T}$
Output: CMFA M, such that $\mathcal{L}(M) = \mathcal{L}(\alpha)$

return *Construct(*$\texttt{root}(\mathcal{T})$*)*

Function *Construct (node t)*
  if $\texttt{type}(t) \in \{[\vee], [\cdot]\}$ **then**
    frLeft $\leftarrow$ *Construct*(*t.leftChild*)
    frRight $\leftarrow$ *Construct*(*t.rightChild*)
    fragment $\leftarrow$ construct automaton fragment by combining *frLeft* and *frRight* according to construction rule for $\texttt{type}(t)$
    fragment.next_counter $\leftarrow$ $max\{$*frLeft.next_counter*, *frRight.next_counter*$\}$
  else if $\texttt{type}(t) \in \{[+], [*], [x\{\}] \mid x \in X\}$ **then**
    frInner $\leftarrow$ *Construct*(*t.child*)
    fragment $\leftarrow$ construct automaton fragment from *frInner* according to rule for $\texttt{type}(t)$
    fragment.next_counter $\leftarrow$ *frInner.next_counter*
  else if $\texttt{type}(t) \in \{[\mathtt{a}], [x] \mid \mathtt{a} \in \Sigma_\varepsilon, x \in X\}$ **then**
    fragment $\leftarrow$ construct automaton fragment according to rule for $\texttt{type}(t)$
    fragment.next_counter $\leftarrow 1$
  else if $\texttt{type}(t) \in \{[\langle a\rangle], [\langle a,b\rangle], [\langle a,\rangle]\}$ **then**
    frInner $\leftarrow$ *Construct*(*t.child*)
    fragment $\leftarrow$ construct fragment from *frInner* according to rule for counting constraint of type $\texttt{type}(t)$ (see Figure 5.2), use counter with number $x =$ *frInner.next_counter*
    fragment.next_counter $\leftarrow$ *frInner.next_counter* $+ 1$
  end
  return *fragment*

Example 5.2. Figure 5.3 shows memory automaton with counters constructed for pattern $(\mathtt{a}\langle 1,4\rangle x_1\{\mathtt{b}^*\})\langle 2\rangle \mathtt{c} x_1(\mathtt{d})\langle 3,\rangle$, the automaton does not entirely follow the construction rules, ε transitions were omitted from the diagram, since they are not important for this example.

The pattern contains three counting constraints. Following rules outlined above, the automaton counters were assigned to them as follows: the nested $\mathtt{a}\langle 1,4\rangle$ constraint is handled by counter number 1, the enclosing $(\ldots)\langle 2\rangle$ is handled by counter 2, and the last constraint $(\mathtt{d})\langle 3,\rangle$, which is independent from the former two, will reuse counter 1. The automaton thus has two counters (and one memory).

The accepted language is $\{a^i b^j a^k b^l c b^l d^m \mid i,k \in [4], j,l \in \mathbb{N}_0, m \geq 4\}$. To illustrate how the CMFA operates, computation on input $w = \mathtt{abaaabbcbbdddd}$

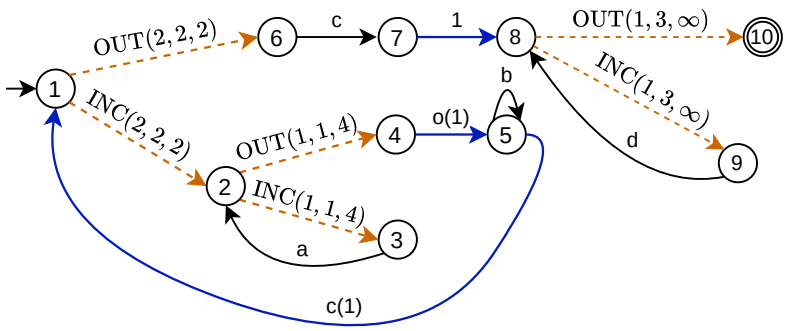


Figure 5.3: An 1-memory automaton with 2 counters (i.e. CMFA(1, 2)) accepting the language of pattern $(\mathtt{a}\langle 1,4\rangle x_1\{\mathtt{b}^*\})\langle 2\rangle \mathtt{c} x_1(\mathtt{d})\langle 3,\rangle$

is shown below. For brevity, the current input string is written as a suffix of input w, the suffix starting at i-th symbol of w is denoted w_i, i.e. it holds $w_i = w[i]\dots w[|w|]$ and $w_1 = w$. Additionally, it is indicated if an input symbol is consumed during transition. The computation runs[27] as follows:

$(1, w_1, (\varepsilon, \mathtt{C}), (0,0)) \vdash (2, w_1, (\varepsilon, \mathtt{C}), (0,1)) \vdash (3, w_1, (\varepsilon, \mathtt{C}), (1,1)) \models^{\mathtt{a}}$
$(2, w_2, (\varepsilon, \mathtt{C}), (1,1)) \vdash (4, w_2, (\varepsilon, \mathtt{C}), (0,1)) \vdash (5, w_2, (\varepsilon, \mathtt{O}), (0,1)) \models^{\mathtt{b}}$
$(5, w_3, (\mathtt{b}, \mathtt{O}), (0,1)) \vdash (1, w_3, (\mathtt{b}, \mathtt{C}), (0,1)) \vdash (2, w_3, (\mathtt{b}, \mathtt{C}), (0,2)) \vdash$
$(3, w_3, (\mathtt{b}, \mathtt{C}), (1,2)) \models^{\mathtt{a}} (2, w_4, (\mathtt{b}, \mathtt{C}), (1,2)) \vdash \dots \models^{\mathtt{a}} (2, w_5, (\mathtt{b}, \mathtt{C}), (2,2)) \vdash \dots \models^{\mathtt{a}}$
$(2, w_6, (\mathtt{b}, \mathtt{C}), (3,2)) \vdash (4, w_6, (\mathtt{b}, \mathtt{C}), (0,2)) \vdash (5, w_6, (\varepsilon, \mathtt{O}), (0,2)) \models^{\mathtt{b}}$
$(5, w_7, (\mathtt{b}, \mathtt{O}), (0,1)) \models^{\mathtt{b}} (5, w_8, (\mathtt{bb}, \mathtt{O}), (0,2)) \vdash (1, w_8, (\mathtt{bb}, \mathtt{C}), (0,2)) \vdash$
$(6, w_8, (\mathtt{bb}, \mathtt{C}), (0,0)) \models^{\mathtt{c}} (7, w_9, (\mathtt{bb}, \mathtt{C}), (0,0)) \models^{\mathtt{bb}} (8, w_{11}, (\mathtt{bb}, \mathtt{C}), (0,0)) \vdash$
$(9, w_{11}, (\mathtt{bb}, \mathtt{C}), (1,0)) \models^{\mathtt{d}} (8, w_{12}, (\mathtt{bb}, \mathtt{C}), (1,0)) \vdash \dots \models^{\mathtt{d}} (8, w_{13}, (\mathtt{bb}, \mathtt{C}), (2,0)) \vdash$
$(9, w_{13}, (\mathtt{bb}, \mathtt{C}), (3,0)) \models^{\mathtt{d}} (8, w_{14}, (\mathtt{bb}, \mathtt{C}), (3,0)) \vdash (9, w_{14}, (\mathtt{bb}, \mathtt{C}), (3,0)) \models^{\mathtt{d}}$
$(8, \varepsilon, (\mathtt{bb}, \mathtt{C}), (3,0)) \vdash (10, \varepsilon, (\mathtt{bb}, \mathtt{C}), (0,0)) \vdash \mathtt{accept}$ ♦

Finally, the matching technique based on active variable degree described in previous section can also be applied for memory automata with counters, because, as also seen on Example 5.2, the function of counting constraints and memories are independent. We implemented both these techniques.

5.2 Solution architecture and implementation details

The implemented regular expression engine was called *mfa-regex*, as it is based on memory automata. The implementation was written in C++ programming language, using the C++14 standard.

[27] There were some non-deterministic choices during the computation, the transition leading to shortest accepting computation was always taken. In practice, the sequence of visited configurations would depend on used matching algorithm.

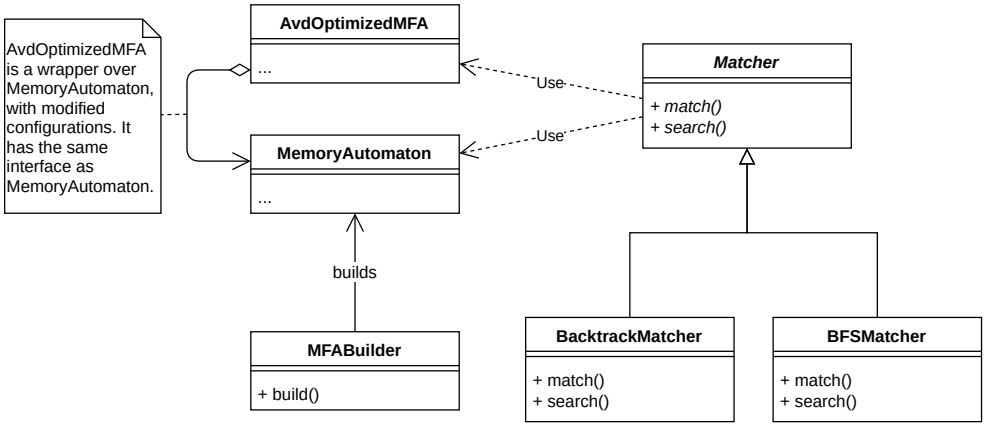


Figure 5.4: Class diagram of the `mfa-regex` implementation

Class diagram illustrating the `mfa-regex` implementation design is presented in Figure 5.4. The implementation consist of three main parts. First is the construction algorithm that converts (compiles) given regex pattern into the memory automaton representation. This is realized by `MFA_Builder` class. It accepts the pattern in form of a string, the string is parsed using a LR parser, or precisely LALR(1), generated by GNU Bison [65] parser generator. The `MFA_Builder` class uses the Bison generated parser to transform the pattern into its syntax tree. Then, memory automaton is constructed for the pattern incrementally from fragments, using the algorithm discussed in Section 4.2 with modifications from Section 5.1 and Section 5.1.5. This is done using a recursive construction function operating on the syntax tree, as illustrated in pseudocode of Algorithm 4.

The representation of memory automaton is the second main part of the solution architecture. Class `MemoryAutomaton` stores the MFA as described in Section 5.1, it holds the graph representation of states and transitions. Additionally, this class implements the transition relation between configurations, function to obtain the initial configuration on an input, and other related functions. However, this class does not implement the matching algorithms, these are separate and form the third main part of the implementation. Configurations are represented as described in Section 5.1.2.

The matching algorithms form a separate class hierarchy, with their interface defined as a C++ abstract class and the concrete algorithms derived from it. As described in Section 5.1.1, BFS and backtracking based matching algorithms were implemented, with the backtracking algorithm having an option to cache all visited configurations and thus behaving as DFS.

Additionally, the efficient matching technique based on active variable degree is implemented by `AvdOptimizedMFA` class. The modified algorithm used

to compute active variables was discussed in Section 5.1.3. The class is implemented as a wrapper over the `MemoryAutomaton` class, having the same interface but using different configurations and a modified transition relation over these configurations, as described in Section 5.1.4. The matching algorithms work independently on which of the automaton classes is used, this is realized using C++ templates.

The `MFA_Builder` class does not need to be used explicitly when constructing memory automata, since the `MemoryAutomaton` (and `AvdOptimizedMFA` too) has constructor accepting regex pattern in form of a string. In this way, the construction process can be transparent when using the library and users only need to deal with memory automata and matcher classes. The library interface is similar to other regular expression implementations. For instance, the `regex` package in Java has `Pattern` class serving as a representation of compiled regular expression, with similar role as our `MemoryAutomaton` class. Then they also have `Matcher` class, which provides the matching functionality.

The *mfa-regex* regular expression engine is provided as a C++ library. It also comes with a command line tool to match given regex pattern and a simple imitation of GNU grep called `mfa-grep`. See Appendix B for the user documentation of *mfa-regex* library.

5.2.1 Supported features

A brief overview of features that are supported by *mfa-regex* is provided in this section along with key details of how they are realized. The complete syntax is described in Appendix B.

The syntax is generally based on POSIX ERE and PCRE syntax. The implementation attempts to support as many of the common extensions found in practical regular expressions as possible, fielding a relatively rich regex syntax.

The standard numbered backreferences, as well as named capture groups and backreferences to them are supported. Unlike most implementations, we also allow capture group content redefinitions (i.e. bounding more than one capture group to the same variable name) using a special syntax. This is done so that the regex patterns from [1] (as per Definition 2.1) are supported. For instance, regex pattern $\alpha = x\{\mathtt{a}^+\}x\mathtt{b}^*x\{\mathtt{c}^+\}x$ would be written as '`(?<x>a+)\k<x>b*(?&x>c+)\k<x>`' in our syntax. The named capture group construct '`(?<x>a+)`' and the '`k<x>`' that is backreference to it are both also part of the PCRE syntax. However, using the same variable name for another capture group would trigger an error in PCRE and most other libraries. Unlike PCRE, we allow to bind variable with another capture group if its name is prefixed with '`&`' in subsequent captures. This behaves the same as variable binding in regex patterns.

For references to capturing groups whose content is not yet defined, we use ε-semantics as established earlier in Chapter 2.

Additionally, counting constraints are supported by implementing the counting memory automata model proposed in Section 5.1.5. We also support character classes and other common constructs.

The library can be compiled with support for Unicode (UTF-8) encoding. We make use of the open source utf-cpp [66] library for this purpose.

The engine also supports searching in input (i.e. finding match anywhere inside the input text), this is done by running the matching algorithm from each position of the input, starting with the first. The left-most match is reported. Start of string and end of string anchors, as well as word boundary assertions are also supported.

Finally, some pattern optimizations were implemented. As this was not the main focus, these are very basic optimizations. Alternation of characters are transformed into character classes where applicable, for example "`a|b|c`" into "`[abc]`". Character classes between alternation terms are also merged where possible, such as "`[abc]|[x0-9]`" into "`[abcx0-9]`". Other simple transformations are performed for quantifiers, such as converting "`(?:a*)*`" into "`a*`".

Optionally, the engine can also convert unused capturing groups into non-capturing groups (i.e. their contents is not stored in memory). We call this optimization *unused memories removal*. During construction, we keep track of which capturing groups (memories) are recalled anywhere in pattern. The capturing groups that are not recalled are then converted into non-capturing ones and number of automaton memories is decreased correspondingly. The k remaining memories are then re-numbered so that their indexes[28] are inside $\{0, \ldots, k-1\}$, naturally all memory recall transitions in automaton must also be changed accordingly to point to the correct memory. However, this optimization is disabled by default, because it limits the provided submatch information.

5.2.2 Testing

To ensure correctness of the implementation, unit tests were developed. The Catch 2 [67] testing framework was used to implement tests. As many as possible cases were tested, with focus on edge cases. All combinations of automaton and matcher classes, as well as possible values of parameters, were tested. The test suite checks several thousands assertions for the different combinations of matching algorithms and other parameters.

Additionally, during experimental evaluation, the results of matching tested inputs on patterns were compared against outputs given by other regular expression engines to check consistency with established implementations.

[28] We number memories from 0 internally, but this is just an implementation detail.

5.2.3 Source code publication and licensing

Source codes of the implemented regular expressions library and tools are enclosed with this thesis, and they are also available at `https://gitlab.com/hronmar/mfa-regex`. The implementation was published under the MIT license.

We make use of two header-only C++ libraries [66, 67] both published under the Boost Software License 1.0 (BSL 1.0). BSL is very similar license to MIT, with the difference that it is slightly more permissive and does not require preservation of copyright and license notices for binary distributions [68]. These two licenses are compatible.

Chapter 6

Experimental evaluation

This chapter describes experimental evaluation of the implemented `mfa-regex` library. The algorithms available in our implementation are compared. Performance and practical usability of the matching technique based on active variable degree is investigated, and it is compared to the simple unused memories removal optimization. Then, the implemented `mfa-regex` library is compared with other existing regular expression engines on various datasets. Finally, complexity properties and resistance to algorithmic complexity attacks via catastrophic backtracking are explored.

6.1 Methodology

The time complexity of deciding the matching problem was measured. In practice, this means to check if the whole input matches the given regular expression pattern. Most libraries provide some *test* function to perform this, but for cases where only searching is supported, each pattern was wrapped inside '`^`' and '`$`' anchors so that each engine performs the same matching test.

For individual matching, the time execution was measured. For calculating performance on whole datasets of multiple patterns and inputs, the time for each run was divided by the input size, resulting in a measure of "matching speed" in the form of $\frac{\text{time}}{\text{input length}}$, this performance measure can then be averaged over multiple inputs and patterns. The input length is measured in number of characters of the input string, but since all inputs in the datasets contain only `ascii` characters, this coincides with input size in bytes. The used datasets of patterns and inputs are described in Section 6.1.1 below, further details about the datasets are provided in Appendix C.

Execution times were measured using the built-in C++ time measurement capabilities, via the `std::chrono` library. The measured times of tested algorithms were averaged over multiple runs for each input, where possible, the

algorithm was run 10 000 times for each input. However, when catastrophic backtracking occurred for some engines, this was indeed not possible. A lower number of runs was performed in such cases. Before the measured runs, 3 unmeasured "warm-up" runs were performed to reduce the influence of caching on the measured execution times.

The programs used for the experiments were compiled using the g++ 9.3.0 compiler, with `-O3` flag enabled. Versions of the other tested libraries are listed in Section 6.3.

6.1.1 Datasets

The following datasets of regular expression patterns were used for experimental evaluation.

regexlib A set of 6 regular expressions taken from regexlib.com [69].

lingua-franca Set of 40 regular expression patterns chosen randomly from the Regex corpus published as part of [70]. The patterns were picked randomly without replacement, but patterns containing unsupported constructs (such as positive/negative look-ahead and look-behind) were not chosen. The patterns were modified by wrapping them inside the begin and end anchors (as described earlier), and other minor changes were necessary in some cases. The source corpus contains 537 806 patterns extracted from 193 524 software projects [70]. It would be impractical to test the implementations on such large corpus, our sample of 40 patterns covers a wide range of cases and should be sufficient for estimating practical regular expression engines performance.

lingua-franca-backref A set of 20 patterns containing backreferences picked randomly from the same Regex corpus as our previous dataset. The sampling was done in the same way, but only choosing from patterns with at least one backreference. The patterns were modified similarly as in the previous case.

catastrophic-backtrack Set of 20 "dangerous" regular expression patterns that may cause catastrophic backtracking when run on a backtracking engine, collected from various sources.

6.1.1.1 Inputs

For the pattern from `lingua-franca` and `lingua-franca-backref`, input strings were generated randomly using the open source randexp.js [71] tool, which for a given regular expression constructs a random string that matches the pattern. Length of the generated string is limited, quantifiers (such as `*`, `+`) have number of repetitions limited by 100. This way, one matching input was

generated for each pattern. Second input, which does not match, was then generated by changing random characters of the matching input. If possible, only one character was modified. Therefore, we have 2 inputs for each pattern, one that matches and one that does not. For `lingua-franca` that is 80 different runs (for pattern, input pairs) in total, and 40 for `lingua-franca-backref`.

Inputs for patterns from `regexlib` and `catastrophic-backtrack` datasets were constructed manually. For the former, two inputs were provided for each pattern, again one that matches and one that does not. For the latter dataset, "dangerous" inputs were chosen that will potentially result in catastrophic backtracking.

6.1.2 Platform specifications

The experiments were run on a system with the following characteristics:

Operating system:	Arch Linux (Linux kernel version 4.9.221)
Processor:	Intel® Core™2 Duo CPU T6570 @ 2.10GHz × 2
Architecture:	64 bit
Memory size (RAM):	4 GB
L2 cache size:	2048 kB
Compiler:	g++ 9.3.0

6.2 Comparison of algorithms and options provided by mfa-regex

This section provides comparison between the different matching algorithms available in our implementation. Then performance of the matching technique based on active variable degree is evaluated.

6.2.1 Comparison of matching algorithms

Performance of the three implemented algorithms was compared, as described earlier in Section 5.1.1, these are the algorithm based on recursive backtracking, DFS (i.e. backtracking with caching every configuration), and BFS.

To get a basic idea of their performance, the algorithms were first run on the 6 patterns from `regexlib` set. The results can be seen in Table 6.1. Each pattern was matched against two input strings, the first one matching and the second one causing a mismatch. Execution times in milliseconds and relative performances are shown. On each of the tested patterns and inputs, backtracking was the fastest from measured matching methods. The additional caching of all visited configurations contributes to the slow down in case of the DFS algorithm.

Table 6.1: Comparison of matching algorithms available in mfa-regex on `regexlib` patterns

Pattern	Input	Res.	Backtrack	DFS	BFS
`^.+@[^.].*\.[a-z]{2,}$`	`John.Doe_1234@example.domain.cz`	T	1 (0.086 ms)	1.78 (0.153 ms)	2.55 (0.219 ms)
`^.+@[^.].*\.[a-z]{2,}$`	`John.Doe_1234@example.domain.c`	F	1 (0.098 ms)	1.96 (0.192 ms)	2.10 (0.206 ms)
`^#?([a-f]|[A-F]|[0-9]){3}(([a-f]|[A-F]|[0-9]){3})?$`	`#7171C6`	T	1 (0.048 ms)	1.40 (0.067 ms)	1.56 (0.075 ms)
`^#?([a-f]|[A-F]|[0-9]){3}(([a-f]|[A-F]|[0-9]){3})?$`	`#7171CG`	F	1 (0.049 ms)	1.39 (0.068 ms)	1.47 (0.072 ms)
`^"([^"](?:\\.|[^\\"]*)*)"$`	`"a␣correctly␣\\␣escaped␣\"C-style\"␣string.\n"`	T	1 (0.072 ms)	1.50 (0.108 ms)	3.50 (0.252 ms)
`^"([^"](?:\\.|[^\\"]*)*)"$`	`"incorrectly␣\\escaped␣\"C-style"␣string.\n"`	F	1 (0.111 ms)	1.75 (0.194 ms)	1.77 (0.197 ms)
`^(\d{1,3}'(\d{3}')*\d{3}(\.\d{1,3})?|\d{1,3}(\.\d{3})?)$`	`123'456'789.123`	T	1 (0.075 ms)	1.45 (0.109 ms)	2.36 (0.177 ms)
`^(\d{1,3}'(\d{3}')*\d{3}(\.\d{1,3})?|\d{1,3}(\.\d{3})?)$`	`123'45'6789.123`	F	1 (0.070 ms)	1.47 (0.103 ms)	1.59 (0.111 ms)
`^([a-zA-Z]:|\\[^/\\:*?"<>|]+\\[^/\\:*?"<>|]+)(\\[^/\\:*?"<>|]+)+(\.[^/\\:*?"<>|]+)$`	`\c\example\path\file_123.txt`	T	1 (0.096 ms)	1.49 (0.143 ms)	2.07 (0.199 ms)
`^([a-zA-Z]:|\\[^/\\:*?"<>|]+\\[^/\\:*?"<>|]+)(\\[^/\\:*?"<>|]+)+(\.[^/\\:*?"<>|]+)$`	`\c\example\path\file_123.`	F	1 (0.108 ms)	1.59 (0.172 ms)	1.59 (0.172 ms)
`^<(\w+)(\s(\w*=".*?")?)*((/>)|((/*?)>.*?</\1>))$`	`<example␣attr1="value1"␣attr2="value2"><childNode/></example>`	T	1 (0.135 ms)	1.76 (0.238 ms)	4.18 (0.564 ms)
`^<(\w+)(\s(\w*=".*?")?)*((/>)|((/*?)>.*?</\1>))$`	`<example␣attr1="value1"␣attr2="value2"><childNode/></exampl>`	F	1 (0.245 ms)	2.14 (0.525 ms)	2.29 (0.560 ms)

Table 6.2: Comparison of matching algorithms on `lingua-franca` dataset

Algorithm	**Avg.** $\frac{\textbf{time}}{\textbf{input size}}$ **[ms/byte]**	**Relative perf.**
Backtrack	0.00253	1
DFS	0.00357	1.411
BFS	0.00426	1.684

Furthermore, in case of BFS, the manner in which the space of configurations is searched results in the method being significantly slower than the other two. This is because while the backtracking algorithm (and DFS) tries to explore the branch of computation as far as possible and backtracks only when necessary, so that final configuration is reached quickly, BFS explores the space "evenly" and thus a lot of unnecessary configurations are visited. As a result of this, there was not such a large difference for BFS between accepting and rejecting an input on some patterns, as was the case of the other two algorithms. For instance, on the last tested pattern, both backtracking and DFS had significantly larger execution time on the rejected (second) input than on the first, e.g. for backtracking it was 0.135 ms versus 0.245 ms. The rejected input is almost the same as the accepted (first) input with minor modification near the end of the strings, and therefore the two algorithms had to backtrack to a large extent, whereas on the accepted input the match was reported quickly. By contrast, because of how BFS searches the space of configurations, it had insubstantial difference between the two inputs, even being slightly faster on the rejected one (0.564 ms versus 0.560 ms).

Additionally, the algorithms were run on the `lingua-franca` dataset, the measured performances are listed in Table 6.2. As described earlier, computation time divided by input size was used as a measure of performance on individual inputs, and this quantity was averaged over the whole dataset. The results are consistent with the findings on patterns from `regexlib`, the fastest algorithm was again backtracking, while DFS was in average slower by approximately 41%, and BFS was more than 68% slower than backtracking.

In conclusion, there seems to be no reason for choosing other matching algorithm than the one based on backtracking. The other two algorithms are interesting for theory, i.e. providing a bound for asymptotic complexity, but in practice backtracking clearly outperforms them. As shown in Section 6.2.2 below, this also holds for patterns with backreferences. Finally, as presented later in Section 6.3.1, the algorithm was resistant to the tested traditional algorithmic complexity attacks.

Table 6.3: Average automaton construction times on `lingua-franca-backref` dataset

Algorithm	Avg. constr. time [ms]	Relative slowdown
Only basic optimizations	0.0424	1
Unused memories removal	0.0441	1.04
Active variable degree opt.	0.1433	3.38

6.2.2 Active variable degree and optimizing memories

In this section, performance of the matching technique based on active variable degree is investigated. We also attempt to estimate the practical usability of this method. The matching algorithm based on active variable degree is compared to the simpler optimization of removing unused memories (i.e converting un-referenced capturing groups to non-capturing) mentioned in Section 5.2.1, and also to the basic matching algorithm with no memory optimizations.

First, the overhead caused by these optimizations was estimated on the `ligua-franca-backref` dataset. Table 6.3 shows average times taken to construct memory automaton for regex patterns from the dataset, compared between the three variants of memory optimizations. When the algorithm based on active variable degree is used, it is necessary to compute active variables during the pattern compilation phase (as shown in Section 5.1.3), after the automaton structure is constructed. This computation of `avd` causes significant overhead, when it was done, the construction times were more than three times slower in average. However, construction times are usually not as crucial as matching times, because construction complexity does not grow with input string size, which is usually much larger than pattern size, and if the same pattern is used to match larger number of inputs, the construction time contributes minimally to the overall execution time. Conversely, unused memories removal caused minimal slowdown.

Comparison of matching times of the three variants on patterns from `ligua-franca-backref` dataset are shown in Figure 6.1. Only the matching times excluding the automaton construction times are plotted in this case. Because no pattern in the `ligua-franca-backref` dataset contain more than two referenced capturing groups, and the only pattern that has two backreferences to different capturing groups also has `avd` = 2, the optimization based on active variables can not help more than simple removal of unused memories. In case of unused memories removal, the matching times were in average approximately 19% smaller for backtracking algorithm. However, implementation of the technique based on active variables introduced some overhead to the matching algorithm, and this overhead out-weights the speed up gained by removing unused memories on this dataset. In average, the matching times

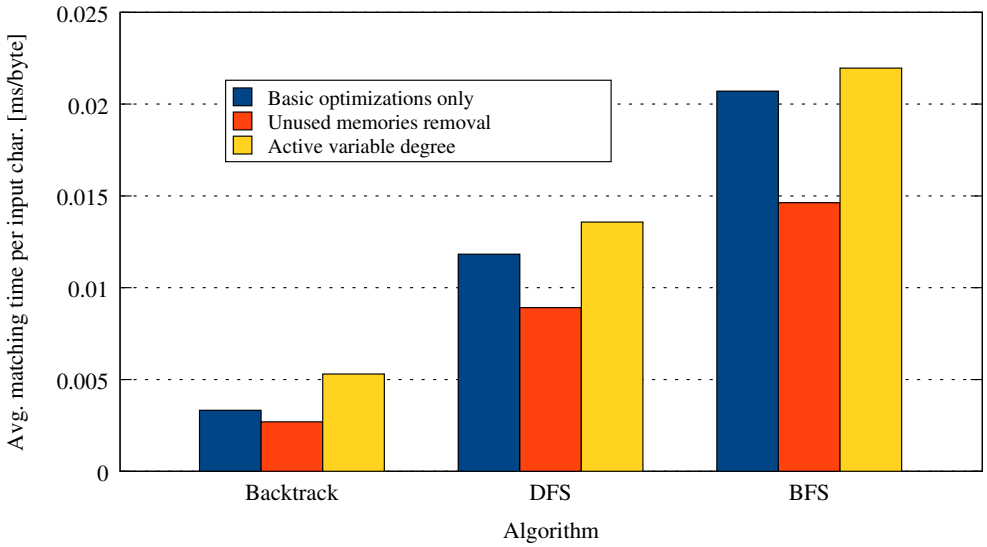


Figure 6.1: Comparison of matching times for different optimizations on `lingua-franca-backref` dataset. Automaton construction times were *not* included in these measurements.

were almost 60% larger in case of backtracking. This could likely be reduced to some extent by optimizing the implementation more.

Additionally, these results further confirm that the backtracking is in practice fastest from the three base matching algorithms, as it had again the smallest matching times for all three variants.

As stated above, the `avd` matching technique could not (even in theory) provided significant improvement on patterns from `ligua-franca-backref` dataset, because the resulting automaton had low number of memories (usually just one). To investigate the practicality of this method and estimate the number of (referenced) capturing groups that must be present in pattern for the method to yield an improved time compared to basic backtracking, we run the following experiment. For $n \in \{1, 2, \ldots\}$, we matched pattern "`.*(.+).*\1.*(.+).*\2....*(.+).*\`n" against an input of the form $a_1^2 a_2^2 \ldots a_n^2$, where $\Sigma = \{a_1 = \texttt{a}, a_2 = \texttt{b}, \ldots\}$, using backtracking with and without the optimization based on active variable degree. It can be seen that the pattern has active variable degree equal to 1, while the number of referenced capturing group in patterns is given by n.

Result of this experiment is shown on Figure 6.2. Execution times were measured, which include both the automaton construction time (plus `avd` computation) and the time spent actually matching the input. We compared the base backtracking algorithm (with only the basic optimizations) and the

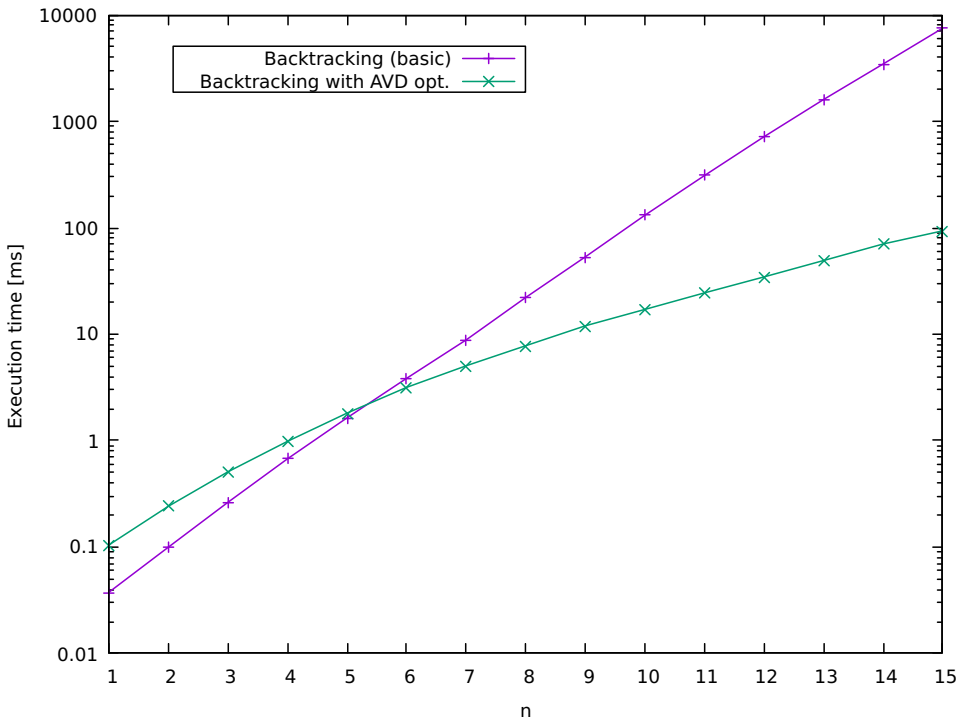


Figure 6.2: Time to match pattern `.*(.+).*\1.*(.+).*\2...*(.+).*\`n on input $a_1^2 a_2^2 \dots a_n^2$, where $\Sigma = \{a_1 = \texttt{a}, a_2 = \texttt{b}, \dots\}$, using backtracking with and without the optimization based on active variable degree.

matching technique based on active variable degree. Note that the simple unused memories removal would have no effect on this pattern, because every capturing group is referenced.

As can be seen on the graph, execution times for the basic backtracking algorithm grow exponentially with the number of backreferences in pattern. In this case, the theoretical worst case described[29] in Section 4.2.1 occurred and the algorithm time complexity was exponentially in the size of the pattern. For larger values of n, the times achieved when the active variable degree optimization was used were significantly lower than in case of simple backtracking. For $n = 15$, the `avd` based algorithm had execution time approximately 93 milliseconds, while the standard backtracking took 7494 milliseconds (i.e. almost 7.5 seconds) to match. These times would likely further grow with pattern or input size. For the tested patterns, `avd` matching outperformed the standard backtracking for number of memories (capturing groups) $n \geq 6$.

[29]Note that the complexity was proven in [2] for searching the space of configurations via BFS. However, DFS has the same asymptotic time complexity, and backtracking is a modified version of DFS.

The usability of the matching technique based on active variables thus depends heavily on the number of capturing groups and backreferences in patterns, and also on the pattern structure. For large number of backreferences to different capturing groups, the method may yield significant speed up compared to simple backtracking. It is a question to what extend are such patterns used in practice. However, the method may still be useful if it is desirable to have better algorithmic complexity upper bound, even if it is paid for by some overhead.

6.3 Comparison with other implementations

The implemented `mfa-regex` library was compared to other existing regular expression engines. For the comparison, backtracking algorithm was used for `mfa-regex`, with only the basic and the unused memories removal optimization. Table 6.4 shows the other libraries (engines) included in the performance comparison and their respective versions.

Table 6.4: Tested libraries and their versions

Library	**Version**
std::regex	GCC 9.3.0 (`libstdc++.so.6.0.28`)
Boost.Regex	Boost 1.72.0
ICU Regular Expressions	ICU 67.1
JPCRE2 (C++ wrapper for PCRE2)	10.31.04 (PCRE2 10.34)
Oniguruma	6.9.5
RE2	2020-05-01

For the PCRE2 library, we used the JPCRE2 [72] C++ wrapper and we did not enable the JIT compilation (this is the default setting in JPCRE2). We also included RE2 library into the comparison, but we note that unlike other tested libraries, RE2 uses different algorithm that is not based on backtracking, see Section 3.5.10 and [61]. RE2's matching algorithm guarantees linear time complexity [61], but this is achieved by abandoning support for backreferences and other features. Because RE2 does not support backreferencing, matching problem for its supported regular expression patterns is in P. Therefore, the comparison with backtracking engines may not be fair, but we included this library to get an idea of how alternative algorithms perform.

The measured execution times of tested libraries on `regexlib` patterns are shown in Table 6.5. Because of space restrictions, inputs for `regexlib` dataset were not listed again in this table (only patterns), see Table 6.1 or Appendix C for list of inputs.

Table 6.5: Comparison of engines on `regexlib` patterns

Pattern	Res.	mfa-regex	std::regex	Boost	ICU	JPCRE2	Onig.	RE2
`^.+@[^.].*\.[a-z]{2,}$`	T	9.56 (0.086 ms)	15.00 (0.135 ms)	1.56 (0.014 ms)	2.67 (0.024 ms)	1.00 (0.009 ms)	1.78 (0.016 ms)	6.33 (0.057 ms)
`^.+@[^.].*\.[a-z]{2,}$`	F	9.80 (0.098 ms)	13.50 (0.135 ms)	1.30 (0.013 ms)	2.40 (0.024 ms)	1.00 (0.010 ms)	1.50 (0.015 ms)	5.40 (0.054 ms)
`^#?([a-f]|[A-F]|[0-9]){3}(([a-f]|[A-F]|[0-9]){3})?$`	T	3.69 (0.048 ms)	30.69 (0.399 ms)	1.77 (0.023 ms)	2.92 (0.038 ms)	1.00 (0.013 ms)	2.00 (0.026 ms)	2.77 (0.036 ms)
`^#?([a-f]|[A-F]|[0-9]){3}(([a-f]|[A-F]|[0-9]){3})?$`	F	3.77 (0.049 ms)	31.54 (0.410 ms)	1.62 (0.021 ms)	3.00 (0.039 ms)	1.00 (0.013 ms)	2.00 (0.026 ms)	2.77 (0.036 ms)
`^"([^"](?:\\.|[^\\"]*)*)"$`	T	7.20 (0.072 ms)	13.60 (0.136 ms)	2.10 (0.021 ms)	2.20 (0.022 ms)	1.00 (0.010 ms)	1.70 (0.017 ms)	5.20 (0.052 ms)
`^"([^"](?:\\.|[^\\"]*)*)"$`	F	4.83 (0.111 ms)	— (>10 min)	— (error)	1 (0.023 ms)	382739 (≈8803 ms)	153304 (≈3526 ms)	2.35 (0.054 ms)
`^(\d{1,3}'(\d{3}')*\d{3}(\.\d{1,3})?|\d{1,3}(\.\d{3})?)$`	T	6.25 (0.075 ms)	32.58 (0.391 ms)	2.75 (0.033 ms)	14.58 (0.175 ms)	1.00 (0.012 ms)	3.42 (0.041 ms)	5.83 (0.070 ms)
`^(\d{1,3}'(\d{3}')*\d{3}(\.\d{1,3})?|\d{1,3}(\.\d{3})?)$`	F	7.00 (0.070 ms)	38.60 (0.386 ms)	3.10 (0.031 ms)	17.40 (0.174 ms)	1.00 (0.010 ms)	4.00 (0.040 ms)	6.10 (0.061 ms)
`^([a-zA-Z]:|\\[^/\\:*?"<>|]+\\[^/\\:*?"<>|]+)(\\[^/\\:*?"<>|]+)+(\.[^/\\:*?"<>|]+)$`	T	6.86 (0.096 ms)	22.86 (0.320 ms)	2.86 (0.040 ms)	2.86 (0.040 ms)	1.00 (0.014 ms)	2.07 (0.029 ms)	8.21 (0.115 ms)
`^([a-zA-Z]:|\\[^/\\:*?"<>|]+\\[^/\\:*?"<>|]+)(\\[^/\\:*?"<>|]+)+(\.[^/\\:*?"<>|]+)$`	F	7.20 (0.108 ms)	20.60 (0.309 ms)	2.67 (0.040 ms)	2.93 (0.044 ms)	1.00 (0.015 ms)	1.93 (0.029 ms)	7.40 (0.111 ms)
`^<(\w+)(\s(\w*=".*?")?)*(/>)|((/*?)>.*?</\1>))$`	T	9.00 (0.135 ms)	12.00 (0.180 ms)	1.67 (0.025 ms)	1.40 (0.021 ms)	1.00 (0.015 ms)	2.60 (0.039 ms)	not supported
`^<(\w+)(\s(\w*=".*?")?)*(/>)|((/*?)>.*?</\1>))$`	F	11.14 (0.245 ms)	9.05 (0.199 ms)	1.18 (0.026 ms)	1.00 (0.022 ms)	1.05 (0.023 ms)	1.86 (0.041 ms)	not supported

The fastest from tested libraries on majority of inputs was PCRE2 (via the JPCRE2 wrapper). Other implementations that performed consistently well on most inputs, with one exception that will be investigated later, were Oniguruma and the Boost Regex library. Our implementation performed significantly slower than PCRE2, but it still outperformed the C++ standard `std::regex` library on every input except one.

On the pattern "`^"([^"](?:\\.|[^\\"]*)*)"$`" (6-th row), most backtracking engines exhibited catastrophic backtracking when matching an input "`"incorrectly␣\\escaped␣\"C-style"␣string.\n"`". This was caused by the nested `*` quantifier. The input is eventually rejected, but a large number of backtracking takes place in traditional backtracking engines. Boost Regex engine threw an exception (runtime error) when catastrophic backtracking was detected. Both PCRE2 and Oniguruma libraries have a limit of backtracking that can take place, and for default values of the limits, both libraries interrupted the matching prematurely on this input and reported an error. To estimate time complexity of the algorithms even on such inputs, the limits were disabled for testing. Unfortunately, to the author's knowledge, such option to disable the backtracking limit is not provided by Boost.

Unlike most other backtracking implementations, `mfa-regex` did not exhibit catastrophic backtracking. This is thanks to the selective caching of configurations as described earlier. It seems that one of the advantages of `mfa-regex` could be the resistance to catastrophic backtracking. This will be investigated later in Section 6.3.1.

Additionally, performance of the libraries was measured on `lingua-franca` and `lingua-franca-backref` datasets. The measured average matching times per input character are presented in Figure 6.3. Note that construction times were included. The results are mostly consistent with the `regexlib` benchmark presented above. Catastrophic backtracking did not occur on any input here. The fastest library was again PCRE2. Our implementation was in average cca. 5.5 times slower than PCRE2 on `lingua-franca` patterns and 8.5 times slower on `lingua-franca-backref` dataset. Still, the execution times were relatively comparable to other regular expression engines, and the implementation performance could likely be improved with more optimizations.

6.3.1 Susceptibility to catastrophic backtracking

In this section, we investigate and compare susceptibility of tested backtracking libraries to catastrophic backtracking. Catastrophic backtracking was discussed in Section 3.3.4, it is a situation when the worst case exponential time behavior occurs for backtracking algorithms. This is often caused by patterns with nested quantifiers. One example of pattern and input causing this behavior was seen earlier on the `regexlib` dataset.

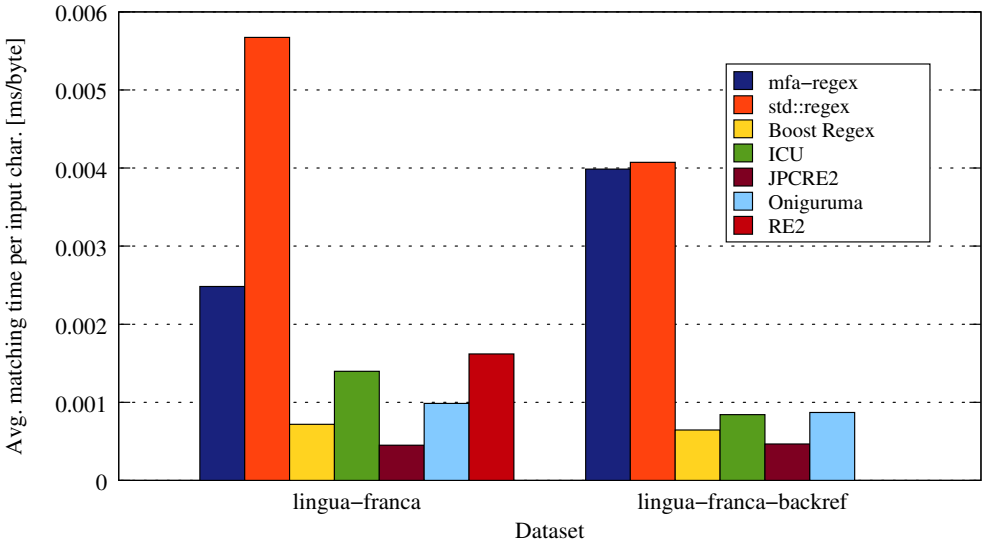


Figure 6.3: Comparison of regular expression engines on `lingua-franca` and `lingua-franca-backref` datasets

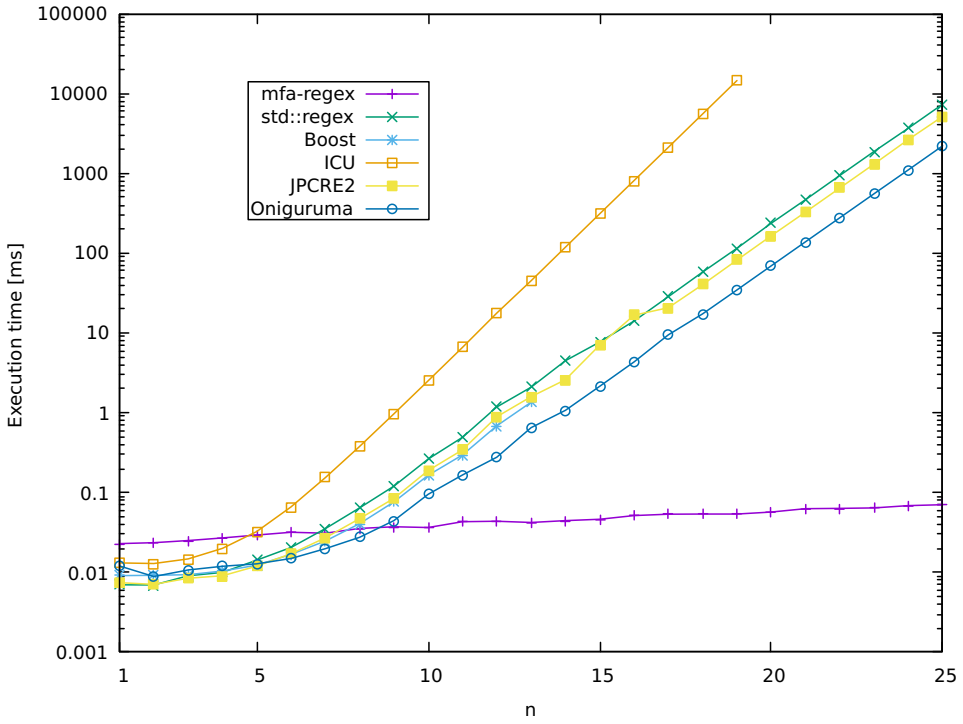


Figure 6.4: Time to match pattern "`(?:.*a)+`" on input $\texttt{a}^n\texttt{b}$

To illustrate how algorithms behave when catastrophic backtracking occurs, the tested backtracking engines were run on pattern "`(?:.*a)+`" and inputs of the form $\texttt{a}^n\texttt{b}$, for $n = 1, 2, \ldots, 25$. The results are presented in Figure 6.4. The matching times of the other backtracking libraries grew exponentially, for $n = 25$ the execution took cca. 2208 milliseconds (i.e. more than two seconds) for the Oniguruma library, and even more for other implementations. Our implementation (`mfa-regex`) did not exhibit exponential behavior in this case, even for $n = 25$, the execution time was 0.07 milliseconds.

As described in Section 4.2.1, matching regex pattern with constant number of backreferences using memory automata has polynomial time complexity. Therefore, in theory, our implementation based on MFA should never exhibit catastrophic backtracking, unless there is a very large number of backreferences to different capturing groups in pattern.

We collected a dataset of 20 patterns and inputs that may potentially lead to catastrophic backtracking. This dataset of "dangerous" regular expressions contains patterns collected from various sources, see Appendix C for details. To test susceptibility of the implementations to catastrophic backtracking, the tested libraries were run on this dataset. If matching took longer than 1000 milliseconds, it was considered a catastrophic backtracking. Note that on inputs of size within few tens of characters, the matching usually took significantly less than one millisecond, when catastrophic backtracking did not occur. The results of this experiment are listed in Table 6.6.

Table 6.6: Susceptibility of libraries to catastrophic backtracking tested on inputs from `catastrophic-backtrack` dataset

Library	**Number of inputs on which catastrophic behavior occurred**
mfa-regex	0
std::regex	19
Boost.Regex	19
ICU Regular Expressions	10
JPCRE2 (PCRE2)	12
Oniguruma	19

Experiments further confirmed the theoretical upper bound for matching complexity when using memory automata with constant number of memories, the `mfa-regex` implementation did not exhibit catastrophic backtracking on any input. From other tested libraries, ICU Regular Expressions and PCRE2 were the most resistant, likely because these engines handle common special cases leading to exponential behavior. However, each of these two libraries still exhibited catastrophic backtracking on at least half of tested inputs. The

remaining three libraries std::regex, Boost, and Oniguruma each exhibited the exponential slowdown on 19 of the tested 20 inputs.

In conclusion, traditional backtracking implementations may provide fastest matching times on most inputs, but in some cases they exhibit exponential matching times. As discussed earlier, this can lead to security vulnerabilities and other problems. Our implementation, on the other hand, provides an upper bound on matching time complexity, as long as the number of backreferences to different capturing groups is limited. Additionally, using the matching technique based on active variables, even patterns with larger number of referenced capturing groups may be matched in reasonable time provided that their active variable degree is low. It may be desirable to have an upper limit on time complexity, even at the cost of slightly increased matching times, especially if potentially dangerous inputs (or patterns) must be processed.

Other existing libraries, such as RE2, also provide an upper bound on matching times. However, they usually achieve this by abandoning support for backreferences and other extensions, as is the case of RE2. Using memory automata as a matching tool provides support for backreferences, and also gives upper bound for the matching time complexity (see Section 4.2.1).

Conclusion

This thesis focused on regular expression matching, especially on the variants of practical regular expressions extended with backreferences. Backreferences were originally invented on purely implementation level, formalization of such patterns and investigation of their theoretical properties was only done later. Chapter 2 provides an overview of the different formalizations established for backreferences and summarizes existing research into properties of such patterns and their language classes. Notably, the matching problem for patterns with backreferences is NP-complete.

Research of existing algorithms and approaches for regular expression matching, with focus on handling the backreferences, is presented in Chapter 3. An overview of existing implementations is also included in the chapter.

As part of this thesis, regular expression library was implemented named *mfa-regex*, as it was based on a computational model of memory automata (or MFA for short) introduced in [1]. First, an overview of this automaton model was given in Chapter 4, along with an outline of its theoretical properties. Then, the recently published matching methods based on memory automata were discussed. We focused in particular on the efficient matching technique based on active variable degree, this algorithm was also implemented into our engine. A relation of the active variable degree property to earlier research on a similar concept is also discussed in the chapter.

Implementation of the regular expression was described in Chapter 5. We first described how the memory automaton model was adapted for practical regular expression matching, the chosen representation of the automaton, and how the construction of MFA from a regular expression pattern was implemented. We implemented the matching algorithm based on breadth-first search of automaton configuration as used in the article [2], and also an alternative algorithm based on recursive backtracking.

Additionally, we implemented the matching method based on active variables from [2]. A new alternative algorithm used by our implementation to compute the active variable sets and active variable degree was described in Chapter 5 and its complexity was examined. We also provided a formalization of the matching algorithm in form of a pseudocode.

Furthermore, we also proposed an extension to memory automaton model for handling counting constraints, an extension commonly found in practical regular expressions. The proposed counters for memory automata were used in our implementation to handle the counting constraints.

Further details about the implementation and the solution architecture were provided in Chapter 5. The implemented `mfa-regex` library was published under the open source MIT license.

Our implementation was compared to other existing regular expression matching tools and the results were presented in Chapter 6. We performed experimental evaluation on datasets of patterns collected from various sources to estimate the practicability and usability of the implemented approaches based on memory automata. Our implementation was among the slower engines on most tested patterns, but it still outperformed the C++ standard *std::regex* library.

However, traditional backtracking engines suffer from an effect called catastrophic backtracking, where on some "dangerous" patterns, an input may trigger exponential worst case matching time. This can happen even for patterns without backreferences. It was discussed that if there is a constant number of referenced capturing groups in pattern (i.e. a constant number of memories in resulting MFA), the matching algorithm based on memory automata has polynomial time complexity. Therefore, unlike the traditional backtracking engines, our implementation (in theory) does not suffer from catastrophic backtracking, unless there is a very large number of backreferences. We confirmed this property on a dataset of "dangerous" patterns, where the other tested libraries that provide support for backreferences each exhibited catastrophic backtracking on at least half of tested inputs. By contrast, for our implementation this behavior never occurred.

The main advantage of matching based on memory automata over the traditional backtracking implementations thus seems to be the time complexity upper bound. Other existing implementations, such as the RE2 library, provide polynomial (or even linear) upper bounds for the matching time complexity. However, this is generally achieved at the cost of not including support for backreferences and other extensions. The implemented matching tool based on memory automata both supports backreferences and provides time complexity upper bound, which is polynomial for constant number of referenced capturing groups. Upper bound on matching time may be useful especially

if the patterns or inputs come from outside sources (e.g. from user), which might pose security risks for the traditional backtracking engines.

Finally, we investigated performance and practicability of the technique based on active variables. We have shown that this method can yield significantly faster matching times on patterns with large number of backreferences to different capturing groups and low active variable degree. However, it is unclear to what extent are such patterns used in practice.

Future work

Apart from the implemented method based on active variable degree, the article [2] presented an additional matching technique for subset of memory automata with a property called memory determinism, which was discussed in Chapter 4. It might be interesting to also implement this method and compare it to the other approaches.

Additionally, during the research of other extensions used in practical regular expressions, we found that subpattern recursion, an extension that further significantly increases the expressive power, is not much researched from the theoretical perspective. It would be interesting to formalize this extension and investigate the exact expressive power of regular expressions extended with both backreferences and subpattern recursion. It seems that the language class of such patterns is a proper superset of context-free languages, but the exact relation with context-sensitive languages is unclear (i.e. whether every context-sensitive language can be expressed). Furthermore, it might be possible to extend memory automaton model to support this feature.

Bibliography

[1] Schmid, M. L. Characterising REGEX languages by regular languages equipped with factor-referencing. *Information and Computation*, volume 249, 2016: pp. 1 – 17, ISSN 0890-5401, doi:10.1016/j.ic.2016.02.003. Available from: `http://www.sciencedirect.com/science/article/pii/S0890540116000109`

[2] Schmid, M. L. Regular Expressions with Backreferences: Polynomial-Time Matching Techniques. *ArXiv e-prints*, 2019, `arXiv:1903.05896`. Available from: `https://arxiv.org/abs/1903.05896`

[3] Freydenberger, D. D.; Schmid, M. L. Deterministic regular expressions with back-references. *Journal of Computer and System Sciences*, volume 105, 2019: pp. 1 – 39, ISSN 0022-0000, doi:10.1016/j.jcss.2019.04.001. Available from: `http://www.sciencedirect.com/science/article/pii/S0022000018301818`

[4] Hopcroft, J. E.; Motwani, R.; et al. *Introduction to Automata Theory, Languages, and Computation*. USA: Addison-Wesley Longman Publishing Co., Inc., third edition, 2006, ISBN 0321462254.

[5] Myhill, J. Linear Bounded Automata. *Wright Air Development Division*, 1960. Available from: `https://ci.nii.ac.jp/naid/10006925353/en/`

[6] Kuroda, S.-Y. Classes of languages and linear-bounded automata. *Information and Control*, volume 7, no. 2, 1964: pp. 207 – 223, ISSN 0019-9958, doi:https://doi.org/10.1016/S0019-9958(64)90120-2. Available from: `http://www.sciencedirect.com/science/article/pii/S0019995864901202`

[7] Kleene, S. C. Representation of Events in Nerve Nets and Finite Automata. In *Automata Studies. (AM-34)*, volume 34, Princeton University Press, 1956, pp. 3–42, doi:10.1515/9781400882618-002.

[8] Aho, A. V.; Hopcroft, J. E. *The Design and Analysis of Computer Algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc., first edition, 1974, ISBN 0201000296.

[9] Aho, A. V. *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*, chapter Algorithms for Finding Patterns in Strings. Cambridge, MA, USA: MIT Press, 1991, ISBN 0444880712, pp. 255–300.

[10] Friedl, J. *Mastering regular expressions*. Sebastopol, CA: O'Reilly, 2006, ISBN 0596528124.

[11] Goyvaerts, J. *Regular expressions cookbook*. Sebastopol, CA: O'Reilly Media, 2012, ISBN 1449319432.

[12] Berglund, M.; van der Merwe, B. Regular Expressions with Backreferences Re-examined. In *Proceedings of the Prague Stringology Conference 2017*, edited by J. Holub; J. Žďárek, 2017, pp. 30–41.

[13] Campeanu, C.; Salomaa, K.; et al. A Formal Study Of Practical Regular Expressions. *Int. J. Found. Comput. Sci.*, volume 14, dec 2003: pp. 1007–1018, doi:10.1142/S012905410300214X.

[14] Carle, B.; Narendran, P. On Extended Regular Expressions. In *Language and Automata Theory and Applications*, edited by A. H. Dediu; A. M. Ionescu; C. Martín-Vide, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, ISBN 978-3-642-00982-2, pp. 279–289.

[15] Schmid, M. L. Inside the Class of REGEX Languages. In *Developments in Language Theory*, edited by H.-C. Yen; O. H. Ibarra, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ISBN 978-3-642-31653-1, pp. 73–84.

[16] Dominus, M. J.; et al. Perl Regular Expression Matching is NP-Hard. Accessed: 2020-02-28. Available from: `https://perl.plover.com/NPC/`

[17] Angluin, D. Finding patterns common to a set of strings. *Journal of Computer and System Sciences*, volume 21, no. 1, 1980: pp. 46 – 62, ISSN 0022-0000, doi:10.1016/0022-0000(80)90041-0. Available from: `http://www.sciencedirect.com/science/article/pii/0022000080900410`

[18] Fernau, H.; Schmid, M. L. Pattern matching with variables: A multivariate complexity analysis. *Information and Computation*, volume 242, 2015: pp. 287 – 305, ISSN 0890-5401, doi:10.1016/j.ic.2015.03.006. Available from: `http://www.sciencedirect.com/science/article/pii/S0890540115000218`

[19] Freydenberger, D. D. Extended Regular Expressions: Succinctness and Decidability. *Theory of Computing Systems (ToCS)*, volume 53, no. 2, 2013: pp. 159–193, doi:10.1007/s00224-012-9389-0.

[20] POSIX.1-2017: The Open Group Base Specifications Issue 7, 2018 edition. IEEE and The Open Group, 2018, accessed: 2020-03-15. Available from: `https://pubs.opengroup.org/onlinepubs/9699919799/`

[21] Popov, N. The true power of regular expressions. 2012, accessed: 2020-03-15. Available from: `https://nikic.github.io/2012/06/15/The-true-power-of-regular-expressions.html`

[22] Cox, R. Implementing Regular Expressions. Accessed: 2020-03-02. Available from: `https://swtch.com/~rsc/regexp/`

[23] Thompson, K. Programming Techniques: Regular Expression Search Algorithm. *Communications of the ACM*, volume 11, no. 6, June 1968: p. 419–422, ISSN 0001-0782, doi:10.1145/363347.363387. Available from: `https://doi.org/10.1145/363347.363387`

[24] Glushkov, V. M. The abstract theory of automata. *Russian Mathematical Surveys*, volume 16, no. 5, oct 1961: pp. 1–53.

[25] Brzozowski, J. A. Derivatives of Regular Expressions. *J. ACM*, volume 11, no. 4, Oct. 1964: p. 481–494, ISSN 0004-5411, doi:10.1145/321239.321249. Available from: `https://doi.org/10.1145/321239.321249`

[26] Cox, R. Regular Expression Matching: the Virtual Machine Approach. Accessed: 2020-03-06. Available from: `https://swtch.com/~rsc/regexp/regexp2.html`

[27] Berglund, M.; Drewes, F.; et al. Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching. *Electronic Proceedings in Theoretical Computer Science*, volume 151, 05 2014, doi:10.4204/EPTCS.151.7.

[28] Dominus, M. J. How Regexes Work. Accessed: 2020-03-02. Available from: `https://perl.plover.com/Regex/article.html`

[29] regex - Henry Spencer's regular expression libraries. Accessed: 2020-03-02. Available from: `https://garyhouston.github.io/regex/`

[30] Schumacher, D. *Software solutions in C*. Boston: AP Professional, 1994, ISBN 9780126323603.

[31] ICU Regular Expressions. Accessed: 2020-05-17. Available from: `http://userguide.icu-project.org/strings/regexp`

[32] Kirrage, J.; Rathnayake, A.; et al. Static Analysis for Regular Expression Denial-of-Service Attacks. In *Network and System Security*, edited by J. Lopez; X. Huang; R. Sandhu, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ISBN 978-3-642-38631-2, pp. 135–148.

[33] Shen, Y.; Jiang, Y.; et al. ReScue: Crafting Regular Expression DoS Attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, New York, NY, USA: Association for Computing Machinery, 2018, ISBN 9781450359375, p. 225–235, doi:10.1145/3238147.3238159. Available from: `https://doi.org/10.1145/3238147.3238159`

[34] Davis, J. C. Rethinking Regex Engines to Address ReDoS. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, New York, NY, USA: Association for Computing Machinery, 2019, ISBN 9781450355728, p. 1256–1258, doi:10.1145/3338906.3342509. Available from: `https://doi.org/10.1145/3338906.3342509`

[35] Becchi, M.; Crowley, P. Extending Finite Automata to Efficiently Match Perl-Compatible Regular Expressions. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, New York, NY, USA: Association for Computing Machinery, 2008, ISBN 9781605582108, doi:10.1145/1544012.1544037. Available from: `https://doi.org/10.1145/1544012.1544037`

[36] Yang, L.; Ganapathy, V.; et al. A novel algorithm for pattern matching with back references. In *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*, 2015, pp. 1–8, doi:10.1109/PCCC.2015.7410264.

[37] Reidenbach, D.; Schmid, M. L. A Polynomial Time Match Test for Large Classes of Extended Regular Expressions. In *Implementation and Application of Automata*, edited by M. Domaratzki; K. Salomaa, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ISBN 978-3-642-18098-9, pp. 241–250.

[38] Shinohara, T. Polynomial time inference of extended regular pattern languages. In *RIMS Symposia on Software Science and Engineering*, edited by E. Goto; K. Furukawa; R. Nakajima; I. Nakata; A. Yonezawa, Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, ISBN 978-3-540-39442-6, pp. 115–127.

[39] Laurikari, V. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *SPIRE 2000*, 2000, ISBN 0-7695-0746-8, pp. 181–187, doi:10.1109/SPIRE.2000.878194.

[40] D'Antoni, L.; Ferreira, T.; et al. Symbolic Register Automata. In *Computer Aided Verification*, edited by I. Dillig; S. Tasiran, Cham: Springer International Publishing, 2019, ISBN 978-3-030-25540-4, pp. 3–21.

[41] Saarikivi, O.; Veanes, M.; et al. Symbolic Regex Matcher. In *Tools and Algorithms for the Construction and Analysis of Systems*, edited by T. Vojnar; L. Zhang, Cham: Springer International Publishing, 2019, ISBN 978-3-030-17462-0, pp. 372–378.

[42] Hopcroft, J. An n log n algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, edited by Z. Kohavi; A. Paz, Academic Press, 1971, ISBN 978-0-12-417750-5, pp. 189 – 196.

[43] Hyperscan 5.2 Developer's Reference Guide. Intel Corporation, 2019, accessed: 2020-03-14. Available from: `http://intel.github.io/hyperscan/dev-reference/`

[44] Pike, R. The Text Editor sam. *SOFTWARE—PRACTICE AND EXPERIENCE*, volume 17, no. 11, 1987: pp. 813–845.

[45] Herczeg, Z. Extending the PCRE Library with Static Backtracking Based Just-in-Time Compilation Support. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, New York, NY, USA: Association for Computing Machinery, 2014, ISBN 9781450326704, p. 306–315, doi:10.1145/2581122.2544146. Available from: `https://doi.org/10.1145/2581122.2544146`

[46] PCRE - Perl Compatible Regular Expressions. Accessed: 2020-03-14. Available from: `https://www.pcre.org/`

[47] pcre2matching man page. Accessed: 2020-03-14. Available from: `https://www.pcre.org/current/doc/html/pcre2matching.html`

[48] The PCRE Open Source Regex Library — Regular-Expressions.info. Accessed: 2020-03-14. Available from: `https://www.regular-expressions.info/pcre.html`

[49] Standard ECMA-262 ECMAScript 2019 Language Specification, 10th edition. ECMA International, 2019, accessed: 2020-04-03. Available from: `https://www.ecma-international.org/publications/standards/Ecma-262.htm`

[50] Corry, E.; Hansen, C. P.; et al. Irregexp, Google Chrome's New Regexp Implementation. Chromium Blog, 2009, accessed: 2020-04-03. Available from: `https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html`

[51] Oniguruma. Accessed: 2020-03-14. Available from: https://github.com/kkos/oniguruma

[52] Shaughnessy, P. Exploring Ruby's Regular Expression Algorithm. 2012, accessed: 2020-03-14. Available from: http://patshaughnessy.net/2012/4/3/exploring-rubys-regular-expression-algorithm

[53] Boost.org regex module. Accessed: 2020-03-14. Available from: https://github.com/boostorg/regex

[54] Berglund, M.; Bester, W.; et al. Formalising Boost POSIX Regular Expression Matching. In *Theoretical Aspects of Computing – ICTAC 2018*, edited by B. Fischer; T. Uustalu, Cham: Springer International Publishing, 2018, ISBN 978-3-030-02508-3, pp. 99–115.

[55] Maddock, J. A Proposal to add Regular Expressions to the Standard Library. 2003, accessed: 2020-03-14. Available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1429.htm

[56] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012, 1338 (est.) pp. Available from: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

[57] The GNU C Library (glibc). Accessed: 2020-03-14. Available from: https://www.gnu.org/software/libc/

[58] Wang, X.; Hong, Y.; et al. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA: USENIX Association, Feb. 2019, ISBN 978-1-931971-49-2, pp. 631–648. Available from: https://www.usenix.org/conference/nsdi19/presentation/wang-xiang

[59] Hyperscan. Accessed: 2020-03-14. Available from: https://github.com/intel/hyperscan

[60] Laurikari, V. TRE. Accessed: 2020-03-14. Available from: https://github.com/laurikari/tre

[61] RE2. Accessed: 2020-03-14. Available from: https://github.com/google/re2/

[62] Aho, A. V.; Lam, M. S.; et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN 0321486811.

[63] Brüggemann-Klein, A.; Wood, D. One-Unambiguous Regular Languages. *Information and Computation*, volume 142, no. 2, 1998: pp. 182 – 206, ISSN 0890-5401, doi:https://doi.org/10.1006/inco.1997.2695. Available from: http://www.sciencedirect.com/science/article/pii/S089054019792695X

[64] Braun, M. moar – Deterministic Regular Expressions with Backreferences. 2016, accessed: 2020-04-04. Available from: https://github.com/s4ke/moar

[65] GNU Bison. The GNU Project, accessed: 2020-04-26. Available from: https://www.gnu.org/software/bison/

[66] Trifunovic, N.; et al. UTF8-CPP: UTF-8 with C++ in a Portable Way. Accessed: 2020-04-28. Available from: https://github.com/nemtrif/utfcpp

[67] Catch2. Accessed: 2020-05-01. Available from: https://github.com/catchorg/Catch2

[68] Dawes, B. Boost Software License. Accessed: 2020-05-25. Available from: https://www.boost.org/users/license.html

[69] RegExLib.com — Regular Expression Library. Accessed: 2020-05-17. Available from: http://regexlib.com/

[70] Davis, J. C.; Michael IV, L. G.; et al. Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-Use and Portability of Regular Expressions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, New York, NY, USA: Association for Computing Machinery, 2019, ISBN 9781450355728, p. 443–454, doi:10.1145/3338906.3338909. Available from: https://doi.org/10.1145/3338906.3338909

[71] randexp.js. Accessed: 2020-05-17. Available from: https://github.com/fent/randexp.js

[72] Hamid, J. JPCRE2. Accessed: 2020-05-22. Available from: https://github.com/jpcre2/jpcre2

Appendix A

Acronyms

AST Abstract syntax tree

avd Active variable degree

avs Active variable set

BFS Breadth-first search

DFA Deterministic finite automaton

DFS Depth-first search

HTTP Hypertext Transfer Protocol

JIT Just-in-time

LBA Linear bounded automaton

MFA Memory automaton

NFA Nondeterministic finite automaton

OS Operating system

PCRE Perl Compatible Regular Expressions

POSIX Portable Operating System Interface

TMFA Memory automaton with trap state

rewbr Regular expressions with backreferences

VM Virtual machine

Appendix B

Documentation for mfa-regex

User documentation of the `mfa-regex` regular expression library implemented as part of this thesis follows. Instructions for building the library are provided. The C++ programming API of the library is documented and the supported syntax is specified. Additionally, usage of the tools included with the library is described.

Building the library

Prerequisites

The project uses CMake (cmake.org) build generator, CMake version 3.1 or newer is required. To compile you need a C++ compiler with support for C++14 standard. GCC (g++) version 9.3.0 and clang version 10.0 are both confirmed to work, but any C++14 compliant compiler should do.

Additionally, GNU Bison (www.gnu.org/software/bison/) *version at least 3.2* is needed for generating the parser. Finally, the Doxygen (doxygen.nl) tool is needed for building the documentation. All the above are standard tools and should also be available as packages on most Linux distributions.

Building and installation

See https://cmake.org/runningcmake/ for instructions on how to use CMake. For example, on Linux the project can be build by running the following commands from the mfa-regex root directory:

```
mkdir build && cd build
cmake ..
make
```

To run the tests, execute:

```
make test
```

Additionally, the documentation may be built by running:

```
make doc
```

Finally, the library and its utilities can be installed using:

```
make install
```

The install path can be chosen during the CMake configuration. Depending on the path, it may be necessary to run the above command as a super user.

Library overview

The mfa-regex library API consists of two main parts: automaton classes, which represent the compiled regex patterns, and matcher classes, which implement the matching algorithms. A brief overview of the API along with examples is provided here, for further details see relevant pages in the Doxygen generated documentation.

The basic automaton class is *MemoryAutomaton*. Its constructor accepts the pattern from which the automaton will be compiled. Second automaton class (with equivalent interface) is *AvdOptimizedMFA*. This class implements the optimization based on active variable degree (see the previously linked article), which has some overhead, but may speed up matching on patterns with large number of capturing groups.

Two matching algorithms are provided: *BacktrackMatcher* and *BFSMatcher*. It is strongly recommended to use the former (BacktrackMatcher), especially if information about the match is required. This is because the breadth-first search algorithm will find *any* match, and may not strictly follow the matching semantics (e.g. greedy, lazy quantifiers). The matches (and submatches) reported by backtracking algorithm will

always follow the matching semantics: the quantifier variants behave as described in the Supported syntax section below, when searching, the leftmost match is returned. The two following functions are provided by the matchers:

Function	Description
match	Attempts to match an entire input on given automaton.
search	Attempts to match any part of input (i.e. performs search) on given automaton.

Both functions return boolean value indicating whether a match was found, and optionally also provide information about the matched substring and contents of the capturing groups. Note that the *match* function returns *true* only if the *whole* input is matched, whereas *search* will report match anywhere inside the input string.

Example code

To illustrate the library usage, the following simple example code is provided. See also how the utilities in src/tools folder are implemented for another examples.

```
#include <mfa-regex/memory_automaton.h>
#include <mfa-regex/backtrack_matcher.h>
#include <iostream>

int main() {
    // create automaton for regex pattern (a+)b*\1
    mfa::MemoryAutomaton automaton("(a+)b*\\1");

    mfa::MatchResult result; // this will store the match details

    /* search for given pattern inside input text "bbaabbaabb"
       using the backtracking algorithm */
    mfa::BacktrackMatcher<mfa::MemoryAutomaton> matcher{};
    if (matcher.search(&automaton, "bbaabbaabb", &result)) {
        std::cout << "found match of length " << result.getLength()
                  << ", starting on position " << result.getPosition()
                  << std::endl;
        std::cout << "matched string: '" << result.getMatchedString() << "'"
                  << std::endl;
        std::cout << "capture group 1 contents: '" << result[1] << "'"
                  << std::endl;
    } else {
        std::cout << "no match" << std::endl;
    }
    return 0;
}
```

Running this program would produce the following output:

```
found match of length 6, starting on position 2
```

```
matched string: 'aabbaa'
capture group 1 contents: 'aa'
```

Additionally, to use the technique based on active variable degree, simply use AvdOptimizedMFA class instead of MemoryAutomaton in the above code (and also include the `avd_optimized_mfa.h` header).

Utilities

Two command-line utilities are included with the library. The first one is `mfa-regex-test`, which is an interactive command-line tool to test regular expression patterns on given inputs, it also presents syntax tree of the pattern for better understanding of what is happening.

The second tool is a (rather minimalistic) imitation of the Unix `grep` command, it is run as:

```
mfa-grep PATTERN [FILE]...
```

Supported syntax

The syntax is based on POSIX ERE and PCRE syntax. Each character in pattern matches itself, except the following special characters (metacharacters):

```
. [ { } ( ) \ * + ? | ^ $
```

To match one of the special characters, prefix it with backslash: for instance `\*` to match '*' as a literal character.

Basic syntax

Token	Description
`*`	Matches the preceding element zero or more times (greedy).
`+`	Matches the preceding element one or more times (greedy).
`?`	Matches the preceding element zero or one times (greedy).
`*? +? ??`	Lazy variants of the three preceding constructs (see below for semantics).
`|`	Alternation, pattern of the form `(a|b)` matches either the a or the b subexpression.
`{k}`	Matches the preceding element exactly k times.
`{k,l}`	Matches the preceding element at least k and at most l times.
`{k,}`	Matches the preceding element at least k times.
`{ }?`	Lazy variant of any of the preceding counting constraints.
`.`	Matches any single character.

Token	Description
`^`	Matches only at the starting position in the input string.
`$`	Matches only at the ending position of the input string.

By default, all quantifiers behave as *greedy*, they always consume as many input characters as possible. For instance, given input "aaab", the pattern `a+` would match "aaa". In contrast, *lazy* quantifiers match as few characters as possible. For the same example, the pattern `a+?` would match only "a".

Bracket expressions

Bracket expression is a construct of the form `[...]` and matches any character between the brackets. For instance `[xyz]` matches 'x', 'y', or 'z'. Bracket expressions can also contain character ranges like `[a-z]`, or character classes `[:class_name:]`. The standard C++ localization library *std::locale* is used for character classes support, see https://en.cppreference.com/w/cpp/header/locale for available character classifications. Additionally, `[:word:]` class is supported and has the same meaning as `[[:alnum:]_]`.

Special characters lose their original meaning inside brackets. Symbol `]` stands for itself only if it is the first character in bracket expression. Similarly, `-` is treated as a literal if it is the first or the last character. `^` matches itself anywhere except the first position inside brackets.

Bracket expression that begins with `^` is *negated*, it matches the complement of characters that a bracket expression without the `^` would match. For example, `[^ab]` matches any characters *except* 'a' and 'b'.

Escape sequences

The following escape sequences have a special meaning. If `\` is not followed by a special character or any of the below escape sequences, the pattern is invalid.

Escape	Description
`\a`	alarm (0x07)
`\e`	escape (0x1B)
`\f`	form feed (0x0C)
`\n`	newline (0x0A)
`\r`	carriage return (0x0D)
`\t`	tab (0x09)
`\Odd`	character with octal code 0dd
`\o{ddd..}`	character with octal code 0ddd..
`\Xdd`	character with hexadecimal code 0xdd
`\X{ddd..}`	character with hexadecimal code 0xddd..
`\C`	any single character (even with Unicode)
`\A`	start of string (like `^`)

Escape	Description
`\Z`	end of string (like `$`)
`\d \l \s \u \w`	same as `[[:digit:]] [[:lower:]] [[:space:]] [[:upper:]] [[:word:]]` respectively
`\D \L \S \U \W`	negated variants of the above
`\b`	word boundary, matches between a character matched by `\w` and a character matched by `\W` or vice versa (also at the beginning or at the end of the string if the first/last character is a word character)
`\B`	not a word boundary (see `\b`)

Grouping and backreferences

Expression	Description
`(...)`	capturing group
`\n`	backreference to n-th capturing group
`\gn \g{n}`	backreference to n-th capturing group
`\g-n \g{-n}`	relative reference to n-th capturing group before the current position
`(?<name>...)`	named capturing group
`(?<&name>...)`	named capturing group redefinition (see below)
`\k<name>`	backreference to named capturing group
`(?:...)`	non-capturing group
`(?|...)`	non-capturing group, reset capturing group index for each alternation term of `...`

Capturing groups are numbered based on the order of their left (opening) parentheses in pattern, starting with 1. Non-capturing groups of any type (such as `(?:...)`) do not participate in this numbering. Non-capturing group of the form `(?|...)` changes the numbering for patterns inside it, each alternative is numbered independently starting with the same number. For instance, in `(?|(a|(b(c)))` both the group `(a)` and `(b)` have number 1, while `(c)` has number 2.

Groups can be referred to by their number, and in case of named groups also by their name. Two groups with different numbers can not be given the same name, in such case redefinition must be used instead. Capture group content is empty by default, if backreference is reached before the corresponding group definition, it matches an empty string.

Unlike most regex engines, mfa-regex supports named capturing group *redefinitions*. Group of the form `(?<&name>...)` has the effect that the content of the group with given name is redefined. For example, the pattern `(?<x>a*)\k<x>(?<&x>b+)\k<x>` first records string matched by `a*` into group named 'x', matches it again, and then it records string matched by `b+` into the same group. Groups of the form `(?<&name>...)`

are not numbered unless there is no preceding `(?<name>...)`, in which case the first `(?<&name>...)` acts as a named capturing group `(?<name>...)` and is numbered accordingly. For example, the pattern `(?:(?<&x>a)|(?<&x>b))\k<x>` is the same as `(?:(?<x>a)|(?<&x>b))\k<x>` and contains one capture group with name 'x' and number 1. Another alternative pattern to this can be constructed using the resetting non-capture group as `(?|(<x>a)|(<x>b))\k<x>`, but capturing groups redefinitions allow to create even more complex patterns. Using both named and numbered backreferences to reference named capturing group is possible but discouraged, since it can lead to confusing patterns.

Appendix C

Datasets used for experimental evaluation

Details about the four datasets that were used for experimental evaluation are provided in this section. The datasets are part of contents of the enclosed CD.

Table C.1 lists each pattern/input pair from `regexlib` dataset, the last column contains `T` (true) or `F` (false) indicating whether given input matches the pattern.

Table C.2 shows first 15 patterns from the `lingua-franca` dataset, the dataset contains in total 40 patterns. Table C.3 contains all patterns from the `lingua-franca-backref` dataset.

Finally, Table C.4 lists all patterns and corresponding inputs that are part of the `catastrophic-backtrack` dataset, including sources from which the patterns were collected (if any).

Table C.1: Patterns and inputs of the `regexlib` dataset

Pattern	Input	Res.
`^.+@[^.].*\.[a-z]{2,}$`	`John.Doe␣1234@example.domain.cz`	T
`^.+@[^.].*\.[a-z]{2,}$`	`John.Doe␣1234@example.domain.c`	F
`^#?([a-f]|[A-F]|[0-9]){3}(([a-f]|[A-F]|[0-9]){3})?$`	`#7171C6`	T
`^#?([a-f]|[A-F]|[0-9]){3}(([a-f]|[A-F]|[0-9]){3})?$`	`#7171CG`	F
`^"([^"](?:\\.|[^\\"]*)*)"$`	`"a␣correctly␣\\␣escaped␣\"C-style\"␣strin g.\n"`	T
`^"([^"](?:\\.|[^\\"]*)*)"$`	`"incorrectly␣\\escaped␣\"C-style"␣string. \n"`	F
`^(\d{1,3}'(\d{3}')*\d{3}(\.\d{1,3})?|\d{1,3}(\.\d{3})?)$`	`123'456'789.123`	T
`^(\d{1,3}'(\d{3}')*\d{3}(\.\d{1,3})?|\d{1,3}(\.\d{3})?)$`	`123'45'6789.123`	F
`^([a-zA-Z]:|\\[^/\\:*?"<>|]+\\[^/\\:*?"<>|]+)(\\[^/\\: *?"<>|]+)+(\.[^/\\:*?"<>|]+)$`	`\c\example\path\file_123.txt`	T
`^([a-zA-Z]:|\\[^/\\:*?"<>|]+\\[^/\\:*?"<>|]+)(\\[^/\\: *?"<>|]+)+(\.[^/\\:*?"<>|]+)$`	`\c\example\path\file_123.`	F
`^<(\w+)(\s(\w*=".*?")?)*((/>)|((/*?)>.*?</\1>))$`	`<example␣attr1="value1"␣attr2="value2"> <childNode/></example>`	T
`^<(\w+)(\s(\w*=".*?")?)*((/>)|((/*?)>.*?</\1>))$`	`<example␣attr1="value1"␣attr2="value2"> <childNode/></exampl>`	F

Table C.2: First 15 patterns of the `lingua-franca` dataset

```
^(.*?):(.*)$
^[0-9]{11}$
^(::)|(127\.0\.0\.1)$
^mysql|maria|sqlite|sqlserver$
^# Filesort:␣(\w+)\s+Filesort_on_disk:␣(\w+)\s+Merge_passes:␣(\d+)$
^(\(|\)|\s|")$
^([0-9]+).*\.sql$
^\w+:␣.*$
^([0-9]+x[0-9]*|[0-9]+)$
^\d\d\d\d\d\d?$
^[a-zA-Z0-9][-a-zA-Z0-9_#␣]*$
^$dps/registrations/PUT/iotdps-register/\?$rid=$
^.*?([0-9]+)\.model\.npz\.SUCCESS$
^<!--\s*LI3_PERF_TOOLBAR\s*-->$
^(-?\d+\.\d+E[-+]?\d+)[FL]?$
```

Table C.3: List of patterns in the `lingua-franca-backref` dataset

```
^(.+?)\1+$
^height=('|")(\d+)\1$
^(\r?\n|\r)\s*\1$
^(/|\\)\.\.?\1$
^source=(['|"])(.*?)\1[␣|}]$
^url\((['"]?)(.*)\1\)$
^\s*mx-diff(?:\s*=\s*(['"])[^'"]+\1)?$
^(['"])\s?\+\s?\1$
^(--|\*{4,}|_{4,}|={4,})(?:\r?\n|\r)(?:.*(?:\r?\n|\r))*?\1$
^(<(h([1-9][0-9]*))[^>]*?>)(.*?)(</\2>)$
^<(filter|macro|typo):([_a-zA-z0-9]+)([^>]*)>(.*?)</\1:\2>$
^(["'])((?:\\\1|.)*?)(\1)$
^x(...)\1\1$
^RELEASE␣+=␣+([\"\'])(\d(\w|\.)+)\1$
^(['"])(.*)\1$
^(@[a-z]+)␣or␣not\(\1\)$
^(VERSION\s*=\s*(["']))\d+\.\d+\.\d+\2$
^[^,]+\.(css|js)(,[^,]+\.\1)*$
^\[(img)\](.*?)\[/\1\]$
^\s*(\d+\.?\d*)\s*(?:;(?:\s*url\s*=\s*(['"]?)(\S*)\2)?\s*)?$
```

Table C.4: Patterns and inputs of the `catastrophic-backtrack` dataset

Pattern	Input	Source
`^(x+x+)+y$`	x^{26}	https://mail.python.org/pipermail/python-dev/2003-May/035916.html
`^(a+)+$`	$a^{29}b$	—
`^(a+)+b$`	a^{40}	https://www.benfrederickson.com/python-catastrophic-regular-expressions-and-the-gil/
`^a?`$^{40}a^{40}$`$`	a^{40}	https://swtch.com/~rsc/regexp/regexp1.html
`^([a-zA-Z]+)*$`	$a^{24}!$	https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS
`^(.*a){20}$`	$a^{24}!$	https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS
`^(.*a)+$`	$a^{24}b$	—
`^([a-zA-Z0-9])(([\-.]|[_]+)?([a-zA-Z0-9]+))*(@){1}[a-z0-9]+[.]{1}(([a-z]{2,3})|([a-z]{2,3}[.]{1}[a-z]{2,3}))$`	$a^{24}!$	http://regexlib.com/REDetails.aspx?regexp_id=1757
`^(([a-z])+.)+[A-Z]([a-z])+$`	$a^{39}!$	https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS
`^(x+x+)+y(?:\1)?$`	x^{30}	—
`^(a+)+b(?:\1)?$`	a^{30}	—
`^A(B|C+)+D(?:\1)*$`	AC^{34}	—
`^(a|aa)+(?:\1)?$`	$a^{34}!$	—
`^(a|a?)+(?:\1)?$`	$a^{24}!$	—
`^(\w+\s?)*$`	(a)	https://medium.com/better-programming/everything-you-need-to-know-about-regular-expressions-in-javascript-59807f758cbd
`^\[(([0-9]*\],\[[0-9]*)*|[0-9]*)\]$`	(b)	https://stackoverflow.com/questions/29751230/regex-pattern-catastrophic-backtracking
`^(a|a?)+(?:\1)?$`	$a^{24}!$	—
`^(A+)*B$`	A^{27}	https://www.rexegg.com/regex-explosive-quantifiers.html
`^(?:[A-Za-z0-9]+[._]?){1,}[A-Za-z0-9]+@(?:(?:[A-Za-z0-9]+[-]?){1,}[A-Za-z0-9]+\.){1,}$`	(c)	http://userguide.icu-project.org/strings/regexp
`^([^b]|a)$`	$a^{49}b$	—

(a) `this␣will␣cause␣catastrophic␣back␣tracking.`

(b) `[1234567],[89023432],[124534543],[4564362],[1234543],[12234567],[124567],[1234567],[1234567]]`

(c) `abcdefghijklmnopq`

Appendix D

Contents of enclosed CD

/
- `readme.txt` ..... the file with CD contents description
- `mfa-regex` ..... directory containing the mfa-regex implementation
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `src` ..... the directory of LaTeX source codes of the thesis
- `benchmarks` ..... directory with files used for benchmarking
  - `measuring` ..... scripts and programs used for the experiments
  - `data` ..... the datasets