BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

BEAT GREP WITH COUNTERS, CHALLENGE

RYCHLEJŠÍ NEŽ GREP POMOCÍ ČÍTAČŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MICHAL HORKÝ**
AUTOR PRÁCE

SUPERVISOR **Mgr. LUKÁŠ HOLÍK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2020/2021

Master's Thesis Specification

Student: **Horký Michal, Bc.**
Programme: Information Technology
Field of study: Information Systems
Title: **Beat Grep with Counters, Challenge**
Category: Algorithms and Data Structures
Assignment:

Regular expressions with repetition, such as ".*a.{999}" (the thousandth character from the end is "a"), is often problematic for regular expression matchers, easily leading to vulnerability to ReDOS attacks. Compilation of regular expressions with repetition into so called counting-set automata has been shown [1] to be a viable alternative to existing matching algorithms that may alleviate the problem. The goal of this work is to implement the method of [1] and evaluate its performance.

1. Familiarise yourself with the work [1] on counting-set automata in pattern matching.
2. Implement an efficient matcher in C++ based on [1].
3. Evaluate its performance against state-of-art matchers, especially on regexes with counting that occur in practice.

Recommended literature:

- Lenka Turonova, Lukas Holik, Ondrej Lengal, Olli Saarikivi, Margus Veanes, and Tomas Vojnar. 2020. Regex Matching with Counting-Set Automata . Proc. ACM Program. Lang. 4, OOPSLA, Article 218 (November 2020), 30 pages. https://doi.org/10.1145/3428286

Requirements for the semestral defence:

- Item 1 of the assignment,
- initiation of the work on item 2,
- at least a small part of the text of the thesis.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Holík Lukáš, Mgr., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 19, 2021
Approval date: November 11, 2020

Abstract

Regular expression matching has an irreplaceable role in software development. The speed of the matching is crucial since it can have a significant impact on the overall usability of the software. However, standard approaches for regular expression matching suffer from high complexity computation for some kinds of regexes. This makes them vulnerable to attacks based on high complexity evaluation of regexes (so-called ReDoS attacks). Regexes with counting operators, which often occurs in practice, are one of such kind. Succinct representation and fast matching of such regexes can be archived by using a novel counting-set automaton. We present a `C++` implementation of a matching algorithm based on the counting-set automaton. The implementation is done within the RE2 library, which is a fast state-of-the-art regular expression matcher. We perform experiments on real-life regexes. The experiments show that implementation within the RE2 is faster than the original `C#` implementation.

Abstrakt

Vyhledávání regulárních výrazů má ve vývoji softwaru nezastupitelné místo. Rychlost vyhledávání může ovlivnit použitelnost softwaru, a proto je na ni kladen velký důraz. Pro určité druhy regulárních výrazů mají standardní přístupy pro vyhledávání vysokou složitost. Kvůli tomu jsou náchylné k útokům založeným na vysoké náročnosti vyhledávání regulárních výrazů (takzvané ReDoS útoky). Regulární výrazy s omezeným opakováním, které se v praxi často vyskytují, jsou jedním z těchto druhů. Efektivní reprezentace a rychlé vyhledávání těchto regulárních výrazů je možné s použitím automatu s čítači. V této práci představujeme implementaci vyhledávání regulárních výrazů založeném na automatech s čítači v `C++`. Vyhledávání je implementováno v rámci RE2, rychlé moderní knihovny pro vyhledávání regulárních výrazů. V práci jsme provedli experimenty na v praxi používaných regulárních výrazech. Výsledky experimentů ukázaly, že implementace v rámci nástroje RE2 je rychleší než původní implementace v jazyce `C#`.

Keywords

regular expression, regex matching, bounded repetition, counting automata, counting-set automata, RE2, `C++`, `C#`

Klíčová slova

regulární výraz, vyhledávání regulárních výrazů, omezené opakování, automat s čítači, automat s čítacími množinami, RE2, `C++`, `C#`

Reference

HORKÝ, Michal. *Beat Grep with Counters, Challenge.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Mgr. Lukáš Holík, Ph.D.

Rozšířený abstrakt

Vyhledávání regulárních výrazů hraje důležitou roli ve vývoji softwaru. Používá se například pro vyhledávání a nahrazování textu, validaci dat, nebo zvýrazňování syntaxe. Na rychlost vyhledávání regulárních výrazů je kladen velký důraz, protože může ovlivnit použitelnost softwaru. Některé přístupy k vyhledávání nemají zaručenou složitost a jejich použití může mít výrazný vliv na používání aplikací. V nejhorším případě může docházet až k nedostupnosti služeb, způsobené nevhodným regulárním výrazem, jehož zpracování má za následek vysoké využívání systémových prostředků. Na vysoké náročnosti zpracování některých regulárních výrazů je založený i útok typu odepření služby (*denial of service*), takzvaný *regular expression denial of service* (ReDoS).

Dnešní moderní nástroje pro vyhledávání regulárních výrazů často používají algoritmy založené na zpětném navracení (tzv. *backtarcking* algoritmy). Algoritmy založené na zpětném navracení vytvářejí z regulárního výrazu *nedeterministický konečný automat* a provádí jeho simulaci. Velkou nevýhodou těchto algoritmů je jejich složitost, která může být v nejhorším případě až exponenciální.

Dalším přístupem k vyhledávání regulárních výrazů jsou algoritmy založené na *deterministických konečných automatech*. Ty mohou být buď předpočítané, nebo se může determinizace provádět v průběhu vyhledávání. Předpočítané automaty mají výhodu v rychlosti vyhledávání, složitost tohoto algoritmu je lineární vzhledem k délce vstupního textu. Nevýhodou je ale právě předpočítávání automatu, u kterého může docházet ke stavové explozi.

Při provádění determinizace v průběhu vyhledávání dochází k simulaci nedeterministického konečného automatu. Tímto způsobem je možné eliminovat riziko stavové exploze. Dnešní moderní nástroje navíc používají cache paměť, kam si postupně ukládají již vypočítané části automatu. Tím částečně řeší problém se složitostí výpočtu v případě, že stavy deterministického automatu jsou tvořeny velkým počtem stavů nedeterministického automatu. Výsledek takového výpočtu mají uložený a nemusí ho tedy provádět znovu. Exploze stavového prostoru je nicméně problematická pro všechny varianty algoritmů založených na konečných automatech.

Častou příčinou stavové exploze jsou regulární výrazy s omezeným opakováním. Tato diplomová práce je založena na publikaci, která představuje automaty s čítači, které dokáží regulární výrazy s omezeným opakování efektivně reprezentovat. Cílem této práce je efektivní implementace vyhledávacího algoritmu založeného na těchto automatech. Práce se také zabývá porovnáním rychlosti výsledného algoritmu s modermíni nástroji a implementací totožného algoritmu v jazyce `C#`.

Automaty s čítači obsahují čítače, které udržují aktuální počet opakování výrazu s omezeným opakováním, například pro regulární výraz `a{1,3}` bude automat obsahovat jeden čítač, který bude udržovat aktuální počet přečtěných znaků `a`. Automat s čítači se pro daný regulární výraz vypočítává pomocí kontrukce založené na tzv. *conditional partial derivatives*.

Automat s čítači je nedeterministický a před jeho použitím pro vyhledávání je nutné provést determinizaci. Tato práce využívá nově představený přístup k determinizaci. Z nedeterministického automatu je vytvářen tzv. *automat s čítacími množinami*. Jedná se o deterministický automat, jehož stavy jsou vybaveny registry, které mohou udržovat množiny celých čísel. Tyto registry jsou používány k simulaci běhu deterministického automatu, kdy za běhu dochází k výpočtu aktuálního stavu paměti jednotlivých čítačů. Jednotlivé přechody automatu s čítacími množinami obsahují tzv. *guards*, jejich splněním je podmíněno provedení přechodu, např. je možné opustit výraz s omezeným opakováním pouze v případě, že bylo dosaženo dolní hranice počtu opakování. Dalším komponentem přechodu jsou

potom operace, které aktualizují paměť registrů při použití přechodu, např. inkrementují hodnotu čítače při přečtění dalšího výskytu symbolu.

Implementace vyhledávacího algoritmu založeného na automatech s čítacími množinami je provedena v rámci nástroje RE2. Jedná se o moderní a rychlý nástroj pro vyhledávání regulárních výrazů obsahující množství optimalizací. Implementace v rámci existujícího nástroje má výhodu v možnosti využití již implementovaných a optimalizovaných částí. Implementace je rozdělena do tří kroků, prvním krokem je vytváření nedeterministického automatu s čítači, druhým krokem je determinizace tohoto automatu a posledním krokem je samotné vyhledávání regulárních výrazů.

V prvním kroce algoritmu se nejprve, za použití existujících funkcí nástroje RE2, provede zpracování vstupního regulárního výrazu. To zahrnuje kontrolu správnosti regulárnho výrazu, zjednodušení některých jeho částí do efektivnější formy a převod regulárního výrazu do interní reprezentace. Součástí tohoto zpracování je i výpočet některých dodatečných informací, jako například tzv. *bytemap classes*, které mapují jednotlivé znaky do skupin, mezi kterými daný regulární výraz nikdy nerozlišuje. Tyto skupiny se potom používají na přechodech automatu. Dalším krokem tohoto bodu algoritmu je normalizace vnitřní reprezentace regulárního výrazu do formy, která odpovídá rovnicím definovaným pro výpočet automatu s čítači. Součástí tohoto kroku je i převod některých operátorů regulárního výrazu na jiné, pro které jsou definovány rovnice. Jedná se o operátor `*` a `?`. Po této úpravě interní reprezentace se regulární výraz postupně prochází zleva, na základě aktuálních podvýrazů se určí správná rovnice a dojde k jejímu výpočtu. Výstupem této části algoritmu je instance třídy `Regexp::Derivatives`, která udržuje nedeterministický automat s čítači. Dále udržuje další informace týkající se tohoto automatu, které jsou potřebné pro jeho determinizaci.

Druhým krokem algoritmu je determinizace automatu s čítači. Jedná se o zobecněnou *subset* konstrukci, která je prováděna podle definovaných formálních rovnic. Nejprve dojde k výpočtu tzv. *scope* čítačů. Tím se zjistí, pro jaké stavy automatu jsou jednotlivé čítače relevantní. S ostními čítači potom není nutné v rámci daného stavu pracovat, protože budou mít vždy implicitní hodnotu nula. V průběhu výpočtu se také doplňují čítače, které patří k operaci **ID** (jedná se o operaci, která nemění stav paměti čítače). Implicitní **ID** operace pro jednotlivé čítače je také vložena na všechny přechody, kde daný čítač není použitý v jiné operaci. Dále je nutné pro jednotlivé přechody získat podmínky pro jednotlivé čítače (*guards*). Nejprve se získají podmínky použitých přechodů nedeterministického automatu pro všechny relevantní čítače (relevantní čítače se určují na základě vypočteného scope). Ze získaných podmínek se vypočtou tzv. *minterms*, jedná se o množinu všech různých kombinací daných podmínek. Pro přechody se ale nepoužijí takové kombinace, které nemohou být nikdy splněny, protože takový přechod by nebylo možné nikdy provést. Pro vytvářený přechod se na základě relevantních čítačů a operací původních přechodů získá množina operací přechodu deterministického automatu. Tato množina operací odpovídá provedení operací všech původních přechodů. Determinizace začíná v počátečním stavu nedeterministického automatu a postupně se prochází všechny nově vytvořené stavy deterministického automatu. Pokud již neexistuje žádný nový stav, je determinizace dokončena.

Posledním krokem algoritmu je samotné vyhledávání regulárního výrazu založeném na automatu s čítacími množinami. Algoritmus provádí determinizaci automatu on-the-fly při samotném vyhledávání. Deterministický automat tedy není předpočítávaný. Vyhledávací algoritmus začíná v počátečním stavu automatu a postupně prochází vstupní text znak po znaku. Pro každý znak si získá jeho bytemap třídu. Následně zjistí, jestli pro danou kombinaci stavu a třídy již byly vypočítané přechody. Pokud ne, provede jeden krok determinizace

pro aktuální stav a třídu, která odpovídá právě zpracovávanému znaku. Tím získá právě používanou část deterministického automatu. Pro danou kombinaci stavu a třídy znaku ale může existovat více přechodů, které se liší v podmínkách čítačů. V průběhu determinizace jsou tyto přechody ukládány do vektoru na různé indexy. Indexy jsou vypočítávané na základě toho, jaký stav paměti může splňovat danou podmínku přechodu. Ve vyhledávacím algoritmu je tedy získán index na základě aktuálního stavu paměti a použije se ten přechod, který je na vypočteném indexu. Při provedení přechodu se aktualizuje paměť čítačů podle operací přechodu a pokračuje se dalším znakem vstupního textu. Po prozkoumání celého vstupního textu se zjišťuje, jestli je poslední navštívěný stav koncový. Pokud je, musí se ještě ověřit, jestli aktuální stav paměti čítačů splňuje koncové podmínky tohoto stavu. Může nastat i situace, kdy koncový stav nemá žádnou podmínku na stav čítačů, v takovém případě je stav koncový bez jakýchkoliv dalších kontrol.

Experimenty prováděné na regulárních výrazech používaných v praxi ukázaly, že implementace v rámci nástroje RE2 je rychlejší než původní implementace v jazyce `C#`. Zrychlení se podařilo dosáhnout především v prvních dvou částech algoritmu, tedy u převodu vstupního regulárního výrazu na automat s čítači a u determinizace tohoto automatu. Pro samotné vyhledávání byla potom implementace v rámci RE2 rychlejší pro většinu regulárních výrazů. Při porovnání s nástrojem grep byla sice implmenetace v RE2 pomalejší pro více regulárních výrazů, průměrný čas vyhledávání nástroje grep byl ale horší. To je způsobeno tím, že pro regulární výrazy s omezeným opakováním (obsahující velký počet opakování) je implementace v rámci RE2 založená na automatech s čítači rychlejší. Obdobných výsledků bylo dosaženo při porovnání s původní implementací nástroje RE2. Jeho původní verze byla oproti novému algoritmu rychlejší ještě ve více případech než grep, ale i zde byly regulární výrazy s omezeným opakováním, pro které byl nově implementovaný algoritmus rychlejší.

Beat Grep with Counters, Challenge

Declaration

I hereby declare that this Diploma thesis was prepared as an original work by the author under the supervision of Mgr. Lukáš Holík Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Michal Horký
May 25, 2021

Acknowledgements

I would like to express my deepest gratitude to my supervisor Mgr. Lukáš Holík, Ph.D., for his patient guidance, invaluable advice, encouragement, and immense knowledge. I would also like to thank Ing. Lenka Turoňová for her advice regarding the implementation and help with the benchmarks.

Contents

Chapter 1

Introduction

Regular expression (*regex*) matching has an irreplaceable role in software development. It is used, for example, for searching, finding and replacing, data validation, or syntax highlighting [48]. As stated in [19, 20, 15], about $30 - 40\%$ of the Python, Java, and JavaScript software uses regular expression matching.

Because not all regex matching engines have complexity guarantees, their efficiency has a significant impact on the performance of the application in which the matching is used. Then, a single poorly written regex can cause excessive CPU use. That could lead to catastrophic consequences, such as the outage of Cloudflare services [27]. Other examples can be the outage of Stack Overflow [3] or a vulnerability in the Express.js [7] framework. The problems are caused by the so-called regular expression denial of service (ReDoS), a denial of service attack based on a high-complexity evaluation of matching regex against a malign text. As stated in works like [19, 20], the ReDoS attack is not just a niche concern but rather a security vulnerability that needs further research.

The cause of the ReDoS attack is a regex that has super-linear worst-case complexity. Such regex can lead to *super-linear behavior* (SL behavior) when the evaluation complexity is polynomial or exponential to the input text length. SL behavior is also known as *catastrophic backtracking*. The catastrophic backtracking is a problem of regex engines that use backtracking-based search algorithms, like the one described by Spencer [45]. *A backtracking regex engine* constructs a *non-deterministic finite automaton* (NFA) from the regex and then simulates the NFA on the input text. Such engines are probably the most implemented ones [20]. An alternative approach is to use *deterministic finite automaton* (DFA), which is pre-computed. This approach, called *static DFA simulation* [42], has much lower worst-case complexity (wrt the length of input text). More precisely, matching can be linear to the length of input text, and each input symbol can be processed in constant time. The major drawback of static DFA simulation is the state explosion of the DFA construction, which can cause significant performance issues when using the method in practice [48].

Another alternative is variants of Thompson's algorithm [46] (also called NFA simulation or NFA-to-DFA simulation). These algorithms work directly with NFA, which results in avoiding the state explosion. The determinization is done *on the fly* by subset construction. The algorithm always remembers only the current DFA state. When reading a character, a next DFA state is computed, and this state is used to replace the current state. The main disadvantage of this approach is that for highly non-deterministic NFA, a set of the NFA states that forms a DFA state can get large. Computing the next DFA state over a symbol then gets expensive, linear to the size of the NFA (in contrast to static

DFA simulation, which does it in constant time). This problem can be partially solved by caching already visited parts of the DFA (a technique used by modern matchers). A step within a cached part is then a constant time operation, the same as for the static DFA simulation. However, regexes that cause exploding determinization are problematic for all variants, explicit determinization, as well as cached or non-cached NFA simulation [48].

This thesis is based on a recent paper by Turoňová et al. [48], which proposes a novel succinct and fast deterministic machine called the *counting-set automaton* (CsA). It is an automaton with so-called *counting sets*, a special type of registers that can hold bounded integer values. It also supports a limited selection of simple set operations. Patterns with the counting operator, also known as the operator of bounded repetition, are a frequent cause of the DFA explosion. Repeated patterns, such as `(ab){1,100}`, can be succinctly expressed by the CsA. Therefore, CsA can eliminate this cause of the DFA explosion [48]. The main goal of this thesis is to implement this novel CsA within state-of-the-art matcher RE2 and evaluate results against the original CsA-based matcher of [48] implemented in `C#` and other state-of-the-art matchers.

This thesis is divided into chapters as follows: Chapter 2 discusses related works and similar algorithms. Chapter 3 contains some basic definitions that will be used throughout the thesis. Chapter 4 firstly overview exact string matching as a simpler matching problem to introduce some of the basic concepts of matching. Then it introduces regular expression matching. This Chapter also describes algorithms used both in exact string matching and regular expression matching. Chapter 5 then overviews the state-of-the-art matchers, algorithms they use, and also some of the optimization techniques. Chapter 6 contains information about the novel counting-set automaton. Implementation of this automaton within RE2 is described in Chapter 7. Chapter 8 then contains information about the experimental evaluation of the implementation from Chapter 7. The last Chapter 9 summarizes the results and discusses possible future improvements.

Chapter 2

Related Work

This Chapter focuses on existing studies focusing on regexes and their derivatives, regexes and automata with counting, and pattern matching of regexes with counting. Apart from introducing the novel counting-set automaton, the paper mentioned above by Turoňová et al. [48] also includes the implementation of a `C#` prototype called `CA`[1] and talks about the experimental evaluation of this prototype. Except for the `C#` language, these are the goals of this thesis. Therefore, the paper by Turoňová et al. [48] is the main related work of this thesis, and, as the related works are very similar, this entire Chapter will be based on the Related work chapter of that paper.

Regexes and Their Derivatives

Efficient matching [22, 37] and match generation [40] can be done using Brzozowski derivatives [11], which provide a practical approach to create a DFA from a regex incrementally. Berry and Sethi [9] were the first to investigate efficient determinization based on Brzozowski derivatives. Construction of NFA from regex can be done using Antimirov derivatives [6] in classical settings. The set $\{D \mid \langle \boldsymbol{ID}, D\rangle \in \partial_a(R)\}$, computed using conditional derivatives equations (Equation 6.8–6.12) without counting loop, is the same as Antimirov derivative of R for a. Generalized Antimirov construction can also be used for extended regexes [13].

Automata With Counting

Holík et al. [29] propose a general determinization of *counting automata* (CAs). It has the same worst-case complexity as the naive explicit determinization, which depends on the set of counters C and the maximum counter upper bound K with factor $(K+1)^{|C|}$, but it can produce smaller automata than the naive explicit determinization. For the class of monadic regexes (single-state-scoped counters and counting on self-loops only), the paper proposes a more efficient algorithm, but it can still generate $(K+1)^{|C|}$ states. It neither talks about derivative construction for translating regexes into CAs nor the application of CA in pattern matching.

Björklund et al. [10] also focus on the use of counters for regexes with bounded repetition. It builds on the formalism of counter automata from [24], called *CNFAs*. A CA from [48] used in this thesis is basically a symbolic generalization of a CNFA, with small technical differences. One of the differences is caused by the usage of a generalized Antimirov construction of CAs, as opposed to generalized Glushkov construction used for

[1] Available at https://pajda.fit.vutbr.cz/ituronova/countingautomata

CNFAs. These constructions are algorithmically quite different. That difference results in counters being 0-based in CAs and 1-based in CNFAs. Björklund et al. [10] focus mostly on a different problem and deterministic regexes. More precisely, the problem is called incremental matching in the context of database queries. It uses a variant of Thompson's algorithm for standard matching. The algorithm is applied directly on CA instead of an NFA. Although in this way, the translation of the regex to an automaton is not dependent on the counter bounds, processing of each character has the same cost as the original Thompson's algorithm (i.e., worst linear to the size of the NFA and the counter bounds).

Matching regexes with counting can also be done using dynamic programming. Kilpeläinen and Tuhkanen [32] introduce a matching algorithm based on dynamic programming. The complexity of this algorithm is at worst quadratic to the length of the input text (in contrast to the linear to the input text length complexity of the determinization and NFA-simulation-based algorithms). The experimental comparison of the variant of Thompson's algorithm used in [10] suggests that the algorithm proposed in [32] is not competitive in practice.

Classical automata can be extended with scratch memory of bits that can represent counter. It is introduced in [43, 44] and called Extended FAs (XFAs). Compilation of regexes into deterministic XFAs consists of two steps; in the first step, an extended version of Thompson's algorithm is used. The second step is done by using an extended version of the classical powerset construction and minimization. A small XFA may exist, but there could be an exponential blowup of the search space arising from determinization for inputs like `.*a.{k}`.

Other automata related to the CAs are R-automata [4]. R-automata operates on a finite number of unbounded counters, but the values of the counters can not be tested. There are also extended finite state machines that are not suitable for the problem of pattern matching considered in this thesis and in [48]. Such machines, which expressive power goes beyond regular languages, can be found, for example, in [8, 17, 41, 44].

Regexes With Counting

Automata with counters, close to CAs used in this thesis, are introduced in [30] and are called FACs. Unlike CAs from [48], they do not allow symbolic character predicates and have fewer kinds of counter updates. Hovland [30] also proposes a conversion from regexes to FACs; it uses a variant of Glushkov automata [25] along with the *first-last-follow* construction [2, 12]. As said in [48], for purposes of the paper, the Antimirov-derivative-based construction provides benefits, such as the generation of one counter per distinct counter sub-expression rather than one per counter position in the regex abstract syntax tree, which results in fewer counters overall. The Antimirov-derivative-based construction was also easier to implement. The Antimirov automaton is in general smaller than the Glushkov automaton. In fact, the Antimirov automaton is a quotient of the Glushkov automaton [14, 31]. Another generalization of Antimirov derivatives, but unrelated to counters, can be found in [33].

Pattern Matching of Regexes With Counting

Based on the analysis of 537k real-world regexes (obtained from a study by Davis et al. [21]) done in [48], the counting operator often appears in regexes in practice, as it was contained in over 33k of the real-world regexes. The .NET ecosystem has regex matchers with two different approaches. The first is based on a backtracking search and is provided in

`System.Text.RegularExpressions`. The second can deal more efficiently with the counting operator. It provides a backtracking-free search without an explicit conversion into a DFA, based on the so-called *symbolic derivatives*. It is the *Symbolic Regex Matcher* (SRM) [40]. The approaches of the extremely optimized state-of-the-art matchers *GNU grep*[2] and *RE2*[3] will be discussed in more detail in Chapter 5.

[2] https://www.gnu.org/software/grep/
[3] https://github.com/google/re2

Chapter 3

Preliminaries

Algorithms for exact string matching and pattern matching described in this thesis are based on automata theory. Although various types of automata are used in pattern matching, this Chapter will introduce basic types of automata. It is also necessary to define regexes, which are used in pattern matching. The definitions in subchapters 3.1 and 3.2 are taken from [52], and the rest (except byte classes) is taken from [48].

3.1 Languages

Before a language can be defined, it is first necessary to define an alphabet and a word.

Definition 3.1.1. *An alphabet is a non-empty set of elements called symbols of the alphabet.*

Definition 3.1.2. *A word (also a string) over the alphabet is every finite sequence of the alphabet symbols. An empty sequence of symbols is called an empty word and is denoted by ϵ.*

Definition 3.1.3. *Given an alphabet Σ, the set of all words over the alphabet is denoted by Σ^*. Set of all non-empty words over the alphabet is denoted by Σ^+ (i.e. $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$). A set L is called a language over the alphabet Σ, if a condition $L \subseteq \Sigma^*$ (or $L \subseteq \Sigma^+$, if the empty word ϵ do not belong to the language) holds. Therefore, a language can be any subset of words over a given alphabet.*

3.2 Automatons

Definition 3.2.1. *A finite-state automaton (FA) is 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ defined as follows:*

- *Q is a finite set of states,*
- *Σ is a finite input alphabet,*
- *δ is a mapping $Q \times \Sigma \to 2^Q$ called transition function (2^Q is the set of subsets of the set Q),*
- *$q_0 \in Q$ is the initial state,*

- $F \subseteq Q$ *is the set of final states.*

An automaton M is called a *non-deterministic finite-state automaton (NFA)* if $\exists q \in Q\ \exists a\ \in \Sigma\colon |\delta(q,a)| > 1$. On the other hand, if $\forall q \in Q\ \forall a\ \in \Sigma\colon |\delta(q,a)| \leq 1$, the automaton is called a *deterministic finite-state automaton (DFA)*. A deterministic finite-state automaton is often defined as follows:

Definition 3.2.2. *A deterministic finite-state automaton (FA) is 5-tuple* $M = (Q, \Sigma, \delta, q_0, F)$ *defined as follows:*

- Q *is a finite set of states,*
- Σ *is a finite input alphabet,*
- δ *is a mapping* $Q \times \Sigma \to Q$*; it is a partial transition function,*
- $q_0 \in Q$ *is the initial state,*
- $F \subseteq Q$ *is the set of final states.*

Definition 3.2.3. *A configuration of an automaton* $M = (Q, \Sigma, \delta, q_0, F)$ *is a pair* $C = (q, w) \in Q \times \Sigma^*$*. An initial configuration is a pair* (q_0, w) *and a pair* (q, ϵ)*, where* $q \in F$ *is a final configuration. A transition of an automaton* M *is represented by binary relation* $\vdash_M$ *on the set of configurations* C*. For all* $q, q' \in Q$ *and* $w, w' \in \Sigma^*$ *it is defined that* $(q, w) \vdash_M (q', w')$ *applies if and only if* $w = aw'$ *for some* $a \in \Sigma$ *and* $q' \in \delta(q, a)$*. The transitive closure of* $\vdash_M$ *is written as* $\vdash_M^+$*, and the transitive and reflexive closure is written as* $\vdash_M^*$*.*

Definition 3.2.4. *An input word* w *is accepted by finite-state automaton* M *if* $(q_0, w) \vdash_M^* (q, \epsilon), q \in F$*. A language accepted by an automaton* $M, L(M)$ *is a set of all words accepted by* M*:* $L(M) = \{w | (q_0, w) \vdash_M^* (q, \epsilon) \wedge q \in F\}$*.*

3.3 Effective Boolean Algebras

Definition 3.3.1. *An effective Boolean algebra* $\mathbb{A}$ *has components* $(\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ *where:*

- $\mathfrak{D}$ *is a universe of underlying domain elements,*
- Ψ *is a set of unary predicates closed under the Boolean connectives* $\vee, \wedge\colon \Psi \times \Psi \to \Psi$ *and* $\neg\colon \Psi \to \Psi$*,*
- $\bot, \top \in \Psi$ *are the false and true predicates,*
- *values of the algebra are sets of domain elements,*
- *the denotation function* $[\![\_]\!]\colon \Psi \to 2^{\mathfrak{D}}$ *satisfies that* $[\![\bot]\!] = \emptyset$*,* $[\![\top]\!] = \mathfrak{D}$*, and for all* $\varphi, \psi \in \Psi, [\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!], [\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$*, and* $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$

When $[\![\varphi]\!] \neq \emptyset$*, for* $\varphi \in \Psi$*,* φ *is called satisfiable, and it is denoted by* $\boldsymbol{Sat}(\varphi)$*. An element* $x \in [\![\varphi]\!]$ *is denoted by* $x \models \varphi$*.*

3.4 Regexes

For purposes of this thesis, the alphabet for regexes will be 8-bit ASCII as it is the alphabet that RE2 is using. In other words, the 8-bit ASCII will be the character universe $\mathfrak{D}$. It is the set $\{n \mid 0 \leq n < 2^8\}$ of all 8-bit characters represented by their character codes. For example, the set of all upper-case letters $\{n \mid 65 \leq n \leq 90\}$ is denoted by `[A-Z]`. When a character class is made up of an individual symbol, it denotes a singleton set, for example, $[\![@]\!] = 64$. Character classes are, in general, closed under Boolean operations. Character classes can be formed using union, then the character class $[\![$`[0-9]`$]\!]$ can be written as `[0-4]` $\cup$ `[5-9]`. Character classes can also be complemented. For example, the character class `[^0-9]` denotes the set of all non-digit characters.

The set of all character classes is an example of the set Ψ of all *predicates* of Boolean algebra. Checking the *satisfiability* of a predicate $\varphi \in \Psi$ means to decide whether φ denotes a *non-empty* set. Examples can be `[]`, which is unsatisfiable because $[\![$`[]`$]\!] = \emptyset$, and `.` that denotes the *true* predicate because $[\![$`.`$]\!] = \mathfrak{D}$.

Predicates from an effective Boolean algebra *CharClass* of *character classes* are the basic building blocks of regexes. An example of such a class can be a class of digits, denoted by `\d`. The concatenation of words u and v is denoted as $u{\cdot}v$ (often written as uv), and it is lifted to sets. For the word $a{\cdot}w$, $a \in \mathfrak{D}$ is called the head of the word and $w \in \mathfrak{D}*$ its *tail*.

The syntax of regexes, where $\alpha \in \Psi_{CharClass}$ and $m, n \in \mathbb{N}, 0 \leq n, 0 < m, n \leq m$, is defined as follows:

$$\epsilon \quad \alpha \quad R_1 \cdot R_2 \quad R_1|R_2 \quad R\{n,m\} \quad R*$$

Furthermore, $\mathcal{L}^n$ denotes the *n-th* power of language $\mathcal{L} \subseteq \Sigma^*$ with $\mathcal{L}^0 \stackrel{def}{=} \{\epsilon\}$ and $\mathcal{L}^{n+1} \stackrel{def}{=} \mathcal{L}^n \cdot \mathcal{L}$. The regex $R_1 \cdot R_2$ denotes a *concatenation node*, and $R_1|R_2$ denotes an *alternation node*.

The semantics of a regex R is defined as a subset of $\mathfrak{D}^*$ as follows:

- $\mathcal{L}(\alpha) \stackrel{\text{def}}{=} [\![\alpha]\!]$,
- $\mathcal{L}(\epsilon) \stackrel{\text{def}}{=} \{\epsilon\}$,
- $\mathcal{L}(R_1R_2) \stackrel{\text{def}}{=} \mathcal{L}(R_1) \cdot \mathcal{L}(R_2)$,
- $\mathcal{L}(R_1|R_2) \stackrel{\text{def}}{=} \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$,
- $\mathcal{L}(R\{n,m\}) \stackrel{\text{def}}{=} \bigcup_{i=n}^{m} (\mathcal{L}(R))^i$,
- $\mathcal{L}(R*) \stackrel{\text{def}}{=} \mathcal{L}(R)^*$.

When $\epsilon \in \mathcal{L}(R)$, R is *nullable*. The number of *character-class leaf nodes* of a regex R is denoted by $\#_\Psi(R)$ and is defined as follows:

- $\#_\Psi(\epsilon) = 0$,
- $\#_\Psi(\alpha) = 1$,
- $\#_\Psi(R_1 \cdot R_2) = \#_\Psi(R_1|R_2) = \#_\Psi(R_1) + \#_\Psi(R_2)$,
- $\#_\Psi(R\{n,m\}) = \#_\Psi(R*) = \#_\Psi(R)$.

3.5 Minterms and Byte Classes

Preds(R) will be the set of all predicates from a regex R. And *Minterms(R)* will denote the set of *minterms* of *Preds(R)*. Minterms from the set *Minterms(R)* are non-overlapping predicates that can be taken as a concrete finite alphabet. Every minterm can be understood as a region in the Venn diagram of the predicates in R. It is satisfiable conjunction $\bigwedge_{\psi \in Preds(R)} \psi'$ where $\psi' \in \{\psi, \neg\psi\}$. An example can be regular expression $R =$ `[0-z]{4}[0-8]{5}`, for which the set $Preds(R) = \{$`[0-8], [0-z]`$\}$ and the $Minterms(R) =$ $\{$`[0-8], [9-z], [^0-z]`$\}$. More formally, when $\alpha \in Minterms(R)$, then $\boldsymbol{Sat}(\alpha)$ and $\forall \psi \in Preds(R)$: $[\![\alpha]\!] \cap [\![\psi]\!] \neq \emptyset \Rightarrow [\![\alpha]\!] \subseteq [\![\psi]\!]$. If the set `X` of predicates consists of intervals used in regex (such as `[a-zA-z]`), the number of minterms is linear in $|$`X`$|$. Although the number of minterms of a general set `X` may be exponential, intervals of numbers generate only a linear number of minterms.

As stated in the source code of RE2[1], a `bytemap` maps bytes to *byte classes*. A byte class represents a range of bytes between which the regex never distinguishes. So, similar to the minterms, byte classes are non-overlapping parts of regex. Even though the byte classes could differ from minterms, the principle of it remains the same. To illustrate that, the byte classes for regex $R =$ `[0-z]{4}[0-8]{5}` will be $\{$`[0-8], [9-z], [^0-z]`$\}$. So for this regex, the byte classes and the minterms are the same. But for the regex $R =$ `.*a{1,3}a{1,3}a`, the set of minterms is $Minterms(R) = \{$`a, [^a]`$\}$. In RE2, the byte classes (written as byte ranges) are: `[0-96]` $\cup$ `[98-127]`, `[97]`, `[128-191]`, `[192-193]` $\cup$ `[245-255]`, `[194-223]`, `[224-239]` and `[240-244]`. So there are seven byte classes in comparison to two minterms. However, the byte classes `[0-96]` $\cup$ `[98-127]`, `[128-191]`, `[192-193]` $\cup$ `[245-255]`, `[194-223]`, `[224-239]` and `[240-244]` will always act the same in the context of the regex $R =$ `.*a{1,3}a{1,3}a`. In fact, these six byte classes are the same as minterm `[^a]`. The only difference between these is that the RE2 split it into more byte classes. Besides this difference, the byte classes mean the same as the minterm: all characters except `a`. Then the byte class `[97]` is the same as minterm `a`. In conclusion, the only difference between minterms and byte classes for this regex is that the byte classes are split into more parts. However, their meaning is the same, and therefore the minterms and byte classes can be treated equally.

3.6 Symbolic Automata

Symbolic finite automata (FAs) are a generalization of classical finite automata, whose alphabet is given by an effective Boolean algebra. Formally, FA is defined as follows:

Definition 3.6.1. *FA is a tuple* $A = (\mathbb{I}, Q, q_0, F, \Delta)$*, where:*

- $\mathbb{I}$ *is an effective Boolean algebra called the input algebra,*
- Q *is a finite set of states,* $q_0 \in Q$ *is the initial state,*
- $F \subseteq Q$ *is the set of final states,*
- $\Delta \subseteq Q \times \Psi_{\mathbb{I}} \times Q$ *is a finite set of transitions.*

[1] To be precise, it is in the *dfa.cc* file available at https://github.com/google/re2/blob/master/re2/dfa.cc

Definition 3.6.2. *A run of A from a state p_0 over a word $a_1 \dots a_n$ is a sequence of transitions $(p_{i-1}, \alpha_i, p_i)_{i=1}^n$ with $a_i \in [\![\alpha_i]\!]$; the run is accepting if $p_n \in F$.*

Definition 3.6.3. *The language of a state q, denoted $\mathcal{L}_A(q)$, is the set of words over which A has an accepting run from q.*

Definition 3.6.4. *The language of A, denoted $\mathcal{L}(A)$, is $\mathcal{L}_A(q_0)$.*

Definition 3.6.5. *FA A is deterministic iff for all $p \in Q$ and all transitions (p, α, q) and (p, α', r), it holds that if $\alpha \wedge \alpha'$ is satisfiable, then $q = r$.*

A classical finite automaton can be understood as a special case of FA. In which the basic predicates have singleton set semantics. That means that for each concrete letter a there is a predicate α_a such that $[\![\alpha_a]\!] = \{a\}$.

Chapter 4

Pattern Matching

Some of the concepts or optimizations used in pattern matching approaches are based on exact string matching algorithms. As the exact string matching is a simpler problem than pattern matching, the algorithms themselves can not be directly used in pattern matching. Therefore the understanding of these algorithms is not necessary for pattern matching. However, it can provide a better understanding of some of the optimizations or approaches used in pattern matching. Information about the exact string matching algorithms is in Section 4.1. For more detailed information on the algorithms, the readers can refer to [36, 16].

In the regular expression matching, as the name suggests, a pattern is represented by a regular expression rather than an exact string. Regexes allow describing text. So, unlike the exact string, which always matches only the same string, a single regex can match multiple different strings. Therefore, regular expression matching is more powerful than exact string matching. For example, when the regex is used to find some text in a text editor, a user can find multiple different words, lines, sentences that have something in common just by using a single regex. As regular expression matching is more powerful than exact string matching, it also requires more complicated algorithms, which can have worse time complexity and could suffer from problems mentioned in Chapter 1.

4.1 Exact String Matching

More formally, the exact string matching problem can be defined as follows:
Let Σ be an alphabet. Input will be a text string $T = t_1t_2t_3 \dots t_n$ and a pattern string $P = p_1p_2p_3 \dots p_m$, where $\forall i \in \{1, \dots, n\} : t_i \in \Sigma$ and $\forall j \in \{1, \dots, m\} : p_j \in \Sigma$. The output will be all locations l of the pattern P in the text T, i.e., $T[l + k] == P[k + 1]$, where $0 \leq k < m$ [16].

The description of all the following exact string matching algorithms has been adopted from [16, 36].

Brute Force Algorithm

The basic and most naive algorithm is the brute force algorithm. An input is a pattern of length m and a text of length n. The algorithm tries to match the first character of the pattern and the text, then the second character, and so on, until it matches the whole pattern or the character mismatch occurs. After that, it will move on to the next position of the text and starts again. Figure 4.1 shows an example of a run of the algorithm.

The time complexity of this algorithm is $O(m \times n)$. For example, when searching for $a^{m-1}b$ in a^n, then for each position in the input text, m comparisons are made.

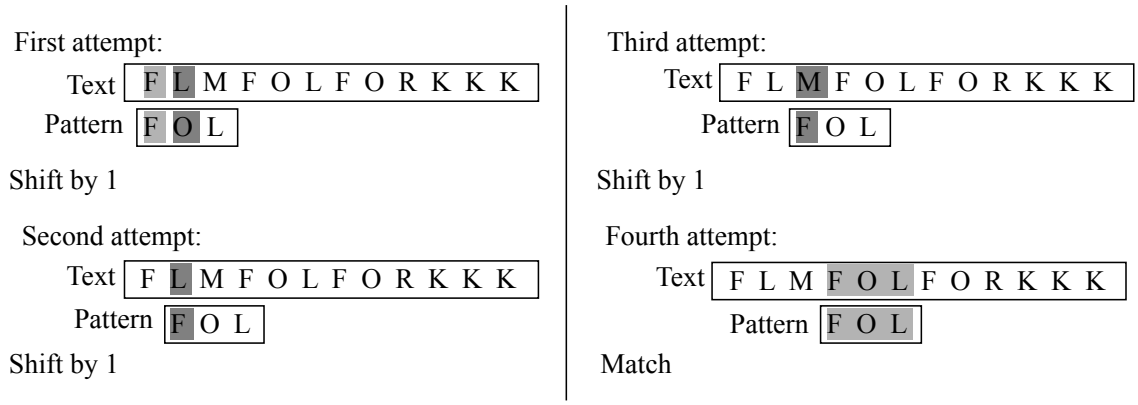


Figure 4.1: Example of a run of the brute force algorithm for exact string matching. The algorithm compares character by character from the first position of text and pattern. It continues to the next position of text and the first position of the pattern after a full match or mismatch. Light gray denotes a successful match, and dark gray denotes mismatch (taken from [16] and edited).

Search With an Automaton

Deterministic Finite Automaton (DFA) $M(x)$ recognizing the language $\Sigma^* x$ is used for searching a pattern x. The first step is to build it.

Example 4.1.1. *The DFA recognizing the language $\Sigma^* x$ is a 5-tuple $= (Q, \Sigma, \delta, q_0, F)$ defined as follows:*

- *Q is the set of all prefixes of x: $Q = \{\epsilon, x[0], x[0..1], \ldots, x[0..m-2], x\}$,*
- *$\Sigma = \Sigma$,*
- *For $q \in Q$ (q is a prefix of x) and $a \in \Sigma$, $(q, a, qa) \in \delta$ if and only if qa is also a prefix of x, otherwise $(q, a, p) \in \delta$ such that p is the longest suffix of qa which is also a prefix of x,*
- *$q_0 = \epsilon$,*
- *$F = \{x\}$.*

The construction of DFA M requires $O(m + \sigma)$ time, and $O(m \times \sigma)$ space, where σ is the size of an input alphabet Σ and m is the size of an input pattern. The searching itself can be performed in $O(n)$ time if DFA is stored in a direct access table. Otherwise, it requires $O(n \times log\sigma)$ time. Here, the size of an input text is denoted by n.

To search a pattern x in an input text, first, the DFA must be built. Then, starting from the initial state q_0, the input text is parsed by $M(x)$. Every time the terminal state is reached, the occurrence of the pattern x is reported. An example of constructed DFA and the run of the algorithm is shown in Figure 4.2.

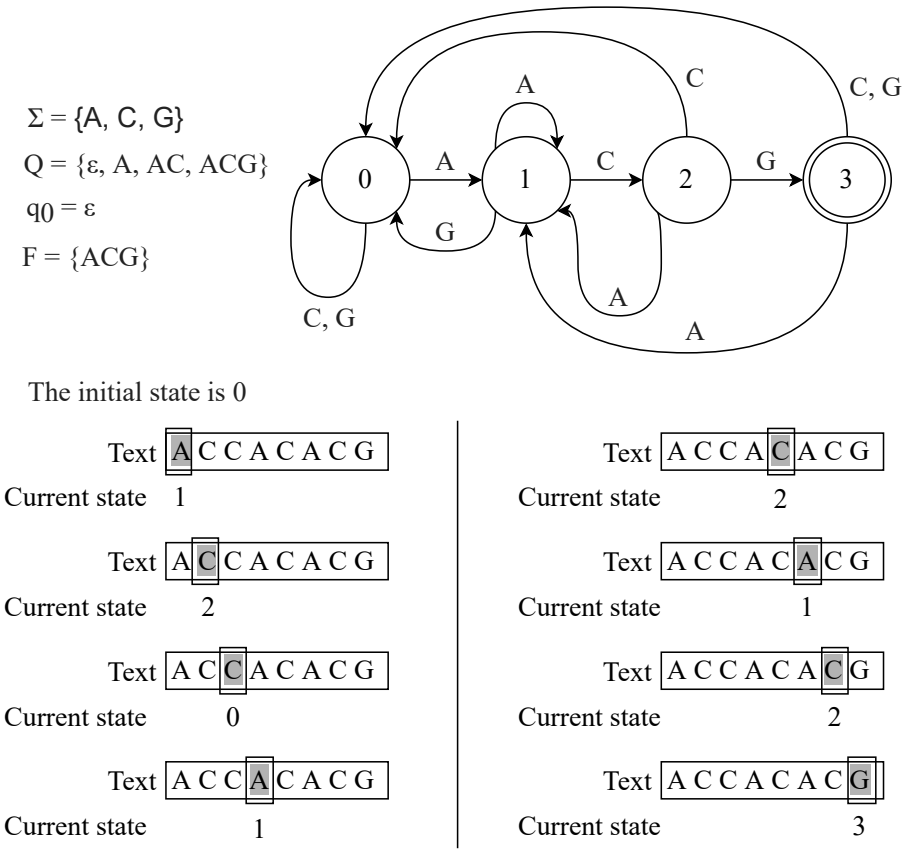


Figure 4.2: Example of the DFA constructed for the pattern "ACG." Labels of the states represent the length of the prefix. The construction is done according to Example 4.1.1. Then, starting from q_0 (in this example, state 0), the input text is parsed by constructed DFA. Each time the final state is reached, the occurrence of the pattern is reported (taken from [16] and edited).

Boyer-Moore Algorithm

The Boyer-Moore algorithm or its modification is often implemented in text editors for the search and substitute command. For usual applications, it is considered the most efficient algorithm.

The algorithm preprocesses the input pattern. The result of preprocessing are two pre-computed functions called *good-suffix shift* (also known as the matching shift) and *bad-character shift* (also known as the occurrence shift). These two functions can be used in case of a mismatch or a complete match of the whole pattern. In both scenarios, the window can be shifted to the right. It is shifted to the right because the algorithm scans the characters of the pattern from right to left, starting from the rightmost one.

Assume that a mismatch occurs between characters at position i in the pattern x and position $i + j$ in the text y (for example, $pattern[i] = a$ and $text[i + j] = b$). Then already scanned part of the pattern and text are the same, so $x[i+1..m-1] = y[i+j+1..j+m-1] = u$ and $x[i] \neq y[i + j]$. The good-suffix shift can be performed in two different ways. The first way is to align the segment $y[i + j + 1..j + m - 1] = x[i + 1..m - 1]$ with its rightmost occurrence in x that is preceded by a character different from $x[i]$ (as shown in Figure 4.3).

The second way of shift is used when there is no such segment. In that case, the longest suffix v of $y[i+j+1..j+m-1]$ is aligned with a matching prefix of x (as shown in Figure 4.4).

The table that stores the good-suffix shift function is called $bmGs$. First, these two conditions must be defined to define the good-suffix shift function:

$$Cs(i,s)\colon \text{ for each } k \text{ such that } i < k < m, s \geq k \text{ or } x[k-s] = x[k] \tag{4.1}$$

$$Co(i,s)\colon \text{ if } s < i \text{ then } x[i-s] \neq x[i] \tag{4.2}$$

Then, for $0 \leq i < m$:

$$bmGs[i+1] = min\{s > 0\colon Cs(i,s) \text{ and } Co(i,s) \text{ hold}\} \tag{4.3}$$

The length of the period of x defines $bmGs[0]$. For the computation of the $bmGs$ table, $suff$ table will be used. The $suff$ table is defined as follows:

$$\text{for } 1 \leq i < m, suff[i] = max\{k\colon x[i-k+1..i] = x[m-k, m-1]\} \tag{4.4}$$

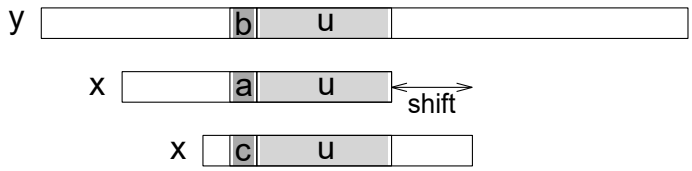


Figure 4.3: The example of the first way of the good-suffix shift. In the pattern x, u re-occurs preceded by a character c that is different from character a (taken from [16]).

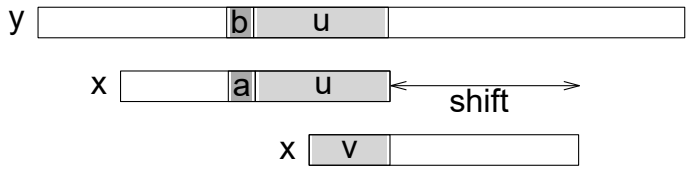


Figure 4.4: The example of the second way of the good-suffix shift. When the first way can not be used, i.e., only the suffix of u re-occurs in x (taken from [16]).

The bad-character shift can also end up in two different ways of shifts. The first is when the character $y[i+j]$ (the mismatched character) occurs in $x[0..m-2]$. Then these two characters are aligned. The second way is used when there is no occurrence of $y[i+j]$ in $x[0..m-2]$. In that case, the character $y[i+j+1]$ (the character immediately after mismatched character) is aligned with the left end of the window (i.e., the left end of the pattern). The bad-character shift can also be negative. The first and the second way of the shifts are in Figure 4.5 and Figure 4.6, respectively.

The table for the bad-character shift function is called $bmBc$ and has the size σ (which is the size of the input alphabet Σ). For $c \in \Sigma$:

$$bmBc[c] = \begin{cases} min\{i\colon 1 \leq i < m-1 \text{ and } x[m-1-i] = c\} & \text{if } c \text{ occurs in } x \\ m & \text{otherwise} \end{cases} \tag{4.5}$$

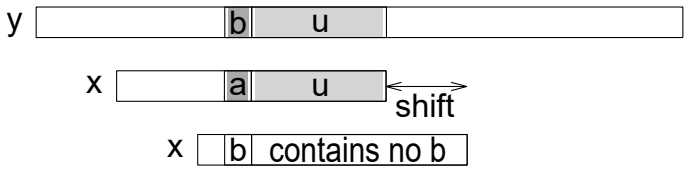


Figure 4.5: The example of the first way of the bad-character shift. The text character $y[i+j]$ occurs in the pattern x. That character is aligned with its rightmost occurrence in x. (taken from [16]).

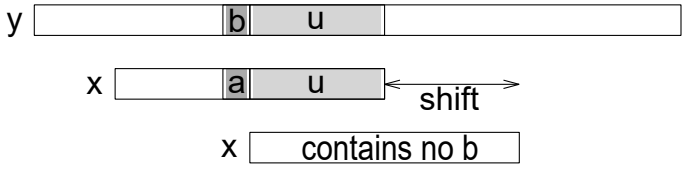


Figure 4.6: The example of the second way of the bad-character shift. The text character $y[i+j]$ does not occur in the pattern x. The left end of the pattern x is aligned with the character $y[i+j+1]$ (taken from [16]).

Both $bmGs$ and $bmBc$ tables can be pre-computed in $O(m+\sigma)$. The pre-computation requires $O(m+\sigma)$ extra space and can be done before the searching phase. The complexity of the searching phase itself is quadratic. But when searching for a non-periodic pattern, a maximum of $3n$ text character comparisons are made. The algorithm is extremely fast on relatively (to the length of the pattern) large alphabets. The algorithm makes only $O(n/m)$ comparisons when searching for the pattern $a^{m-1}b$ in the text a^n. For string-matching algorithms where only the pattern is preprocessed, this is an absolute minimum number of text comparisons.

An example of a run of the Boyer-Moore algorithm is in Figure 4.7.

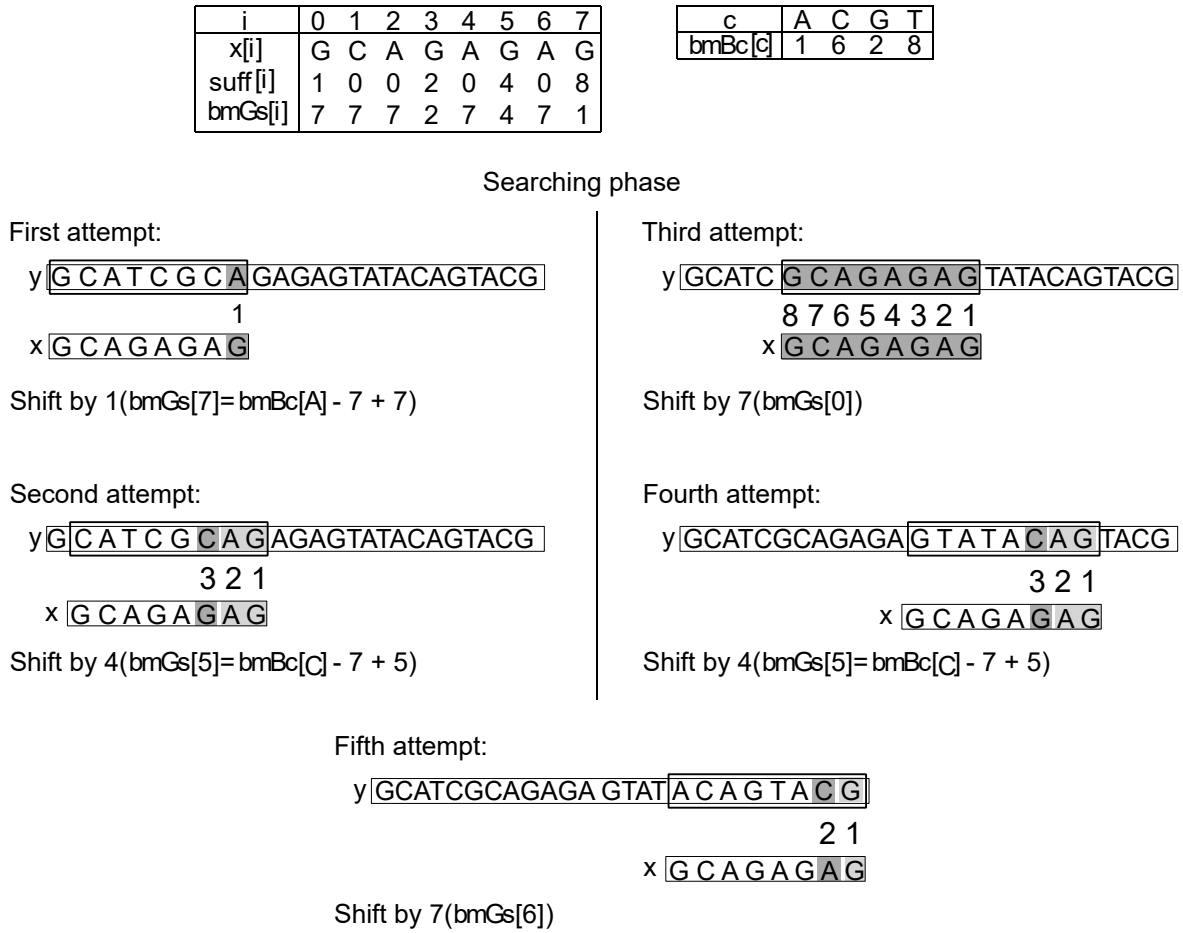


Figure 4.7: Example of the run of the Boyer-Moore algorithm. The good-suffix shift and bad-character shift functions (the *bmGs* and *bmBc* tables) are used to determine the shift of the pattern. In this example, the Boyer-Moore algorithm performs 17 text comparisons. (taken from [16] and edited).

4.2 Regular Expression Matching

Regexes provide extended possibilities to describe a pattern to be searched in a text. In regexes, two types of characters can be used. The special characters (an example of the special character can be "." character, which matches a single character), called *metacharacters*. All other characters are normal text characters, called *literals* [23].

The task is the same as in exact string matching, e.g., to find one, or alternatively all, the occurrences of a pattern in the given text. The difference is that the pattern is a regex. As it provides extended capabilities of describing text, the regex can then match more different strings that have something in common. For example, regex `[A-Z]+` will match all upper case sequences in text, `[A-Z][a-z]{1,5}` will match all strings starting with one upper case letter followed by one to five lower case letters.

Backtracking

Backtracking is used in algorithms that use NFAs. A regular expression is first converted to an NFA, which is then used for matching by the regex engine. An NFA can be in some state from which it can make a transition to more than one state based on a current character in the input text. It will make the transition to one of them and remembers the rest. The other transitions then can take place later if needed. The situation when it has more than one option to make a transition will always happen when there is a quantifier (so it has to decide whether to try another match or continue with the rest of the regex) or alternation (so it has to decide which of the alternatives to try first) in the regex [23].

Independently on the selected transition, if the rest of the match is successful, the whole match is also successful. If the rest of the regex can not be matched after that transition, the whole match still could be successful. Because the regex engine remembers the rest of the possible transitions, it could backtrack to the state before that transition and try one of the other options. This way, the engine will try at least as many different transitions as needed to a successful match. This behavior could eventually lead to trying all possible permutations of the regex [23].

Matching regex `to(nite|knight|night)` on the string `hot tonic tonight` using backtracking will go as follows. First, the regex engine tries to match the first letter of the string, which is `h`, and the first letter of the regex, which is `t`. This attempt will fail, same as the next letter of the string at the second position. Then, on the third position of the string, the letter `t` will match. However, it will fail right on the next position, which is a space in the string and `o` in the regex. The engine also tries to match the letter `t` of the regex against the space in the string, but this will also fail. When the engine reaches word `tonic` in the string, it will successfully match `t` and `o`. Then, the regex provides three different options, which are `nite`, `knight`, and `night`. The engine picks one and remembers the others in case the selected option fails. Assume that the engine picks, for example, `nite` as the first option to try. It will match letters `n` and `i`; then it fails while matching the letter `c` of the string and the letter `t` of the regex. In this case, unlike the first fails, the engine will not shift to the next position in the string. Instead, it will backtrack to the state where it has chosen `nite` from the regex and try the next option. The engine also goes back to the last matched position in the string. So the `to` in word `tonic` is matched, and the engine now tries to match `knight` from the regex. This attempt will fail instantly as the letter `n` from the string does not match the letter `k` from the regex. So the engine will backtrack again and try the last option from the regex, which is `night`. This time, the engine successfully matches the letters `n` and `i` before it will fail. As `night` was the last option, this failure means that the whole attempt starting at word `tonic` fails too. The engine will continue unsuccessful attempts until it reaches the word `tonight` of the string. Then, it will first match the letters `t` and `o`. Then it will fail with the first two options (`nite` and `knight`). Finally, with the last option `night`, the match is found [23].

Backtracking engines could suffer from the problem mentioned in Chapter 1, called catastrophic backtracking. A simple example of a regex that will suffer from catastrophic backtracking can be `(x+x+)+y`, where `x` could represent something more complex. The catastrophic backtracking will not happen on all strings. When the regex is matched against a good string, like `xxxxxxxxxxy`, the matching will be processed without any problem. First, all the ten `x` letters will be matched by the first `x+`. Then, the second `x+` fails while trying to match the letter `y`. As the `x+` must match at least one letter `x`, the engine will backtrack one step back when the first `x` was matching only the first nine letters. In that

state, there is one letter `x` remaining, which will be matched by the second `x+`. When all `x` letters are matched by both `x+`, the group is matched once. The engine will try to match the whole group again, but it will fail right in the first step as there is the letter `y` in the string. However, since one repetition of the group is sufficient, the group matches. The engine will backtrack one step back to the state where all letters `x` are matched. As the last step, the regex engine will match the letter `y`, and the whole match is successful [26].

Problems start to appear when there is no `y` in the input string. Such malign string could be, for example, `xxxxxxxxxx`. The engine will start matching the same as with the first string. At the end of the string, it will fail to match the letter `y`, and it will backtrack. The group has one iteration to backtrack. Since the second `x+` matched only one letter, it can not backtrack. So the first `x+` must give up one letter. Then, the second `x+` will match `xx`. The group has one iteration matched, failing to match the next iteration again. The `y` will also fail. So the regex engine has to backtrack again. However, now, it can backtrack in the second `x+` since it matched `xx`. The match is reduced to one `x`. In the next step, the regex engine tries to match the second iteration of the group. The first `x+` will match, but the second fails at the end of the string. The regex engine has to backtrack again in this step by reducing the match of the first `x+` to seven letters. The second `x+` match `xxx`, then the regex engine fails to match `y`, reducing to `xx` and `x` for the second `x+` in the next steps. The group now can match the second iteration, matching one `x` for each `x+`. However, this attempt will fail too. The engine will continue trying all other possible combinations, all of them failing, since there is no `y` in the string [26].

According to RegexBuddy's debugger[1], matching the regex `(x+x+)+y` against the string `xxxxxxxxxx` (the letter `x` ten times) will take 2558 steps to fail. When the string is extended to eleven letters `x`, twelve letters `x`, and eighteen letters `x`, it will take 5118, 10238, and 655358 steps, respectively, for the regex engine to fail the match. Any string longer than eighteen letters composed of the letter `x` will take over a million steps to fail. Therefore, matching the regex on such malign strings will lead to exponential complexity of $O(2^n)$ [26].

Thompson's Algorithm

Thompson's algorithm was first introduced in the paper by Thompson [46]. This algorithm does not use backtracking. It instead examines the input string character by character against a list of all possible current characters. While the algorithm traverses the list of all possible characters, it simultaneously builds a list of all possible next characters. When the current list is traversed, the newly built list of the next possible characters becomes the current list. Then the next character from the input text is obtained, and the examination continues. Concerning Brzozowski derivatives [11], the algorithm continually takes the left derivative of the regex with respect to the input string. The algorithm is also naturally parallel, which makes it extremely fast [46].

Thompson [46] also introduced a specific implementation of the algorithm, a compiler, which translated a regular expression into `IBM 7094` code. The compiler consists of three parts. The first part checks the syntactical correction of the regular expression. It also inserts the dot operator for the juxtaposition of the regular expression. The regex is then converted to reverse Polish form by the second stage of the compiler. The last, third stage is the object code producer. The third stage uses a pushdown stack, where compiled codes of operands are stored. During the compilation, a unary operator, such as the star operator, works with the top entry of the stack. The result operand replaces the original top of

[1] Available at www.regexbuddy.com

the stack and is available for another operation. Binary operators are compiled similarly. The only difference is that it works with the top two entries of the stack. It also replaces both stack entries rather than only one.

Two functional routines, `NNODE` and `CNODE`, are invoked by the compiled code. `NNODE` matches a single character, and `CNODE` split the current search path. They are used as operands on the stack [46]. Even though the paper by Thompson [46] does not explicitly mention NFA, as stated in [38], the latent NFA construction can be seen in Thompson's algorithm. Furthermore, Cox [38] provides an implementation of Thompson's algorithm, which is not compiling the regex to machine code and is written in C. Also, there is a standard automaton approach to convert the regex into the NFA. The construction is straightforward in the automata theory. The construction algorithm is described in [34]. However, this section talks about Thompson's algorithm, so Figure 4.8, Figure 4.9, Figure 4.10, Figure 4.11, and Figure 4.12 show the functions of the third part of the compiler from the paper by Thompson [46]. Regex `a(b|c)*d`, firstly translated into `abc|*.d.` by the second part of the compiler, is used for the example.

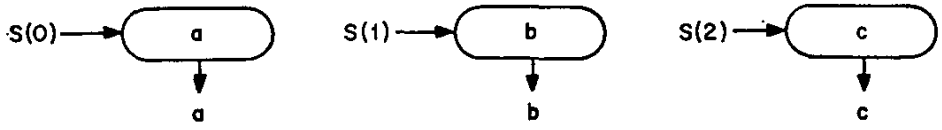


Figure 4.8: Each of the three characters, `a`, `b`, and `c`, creates a stack entry and an `NNODE`, which match a single character (taken from [46]).

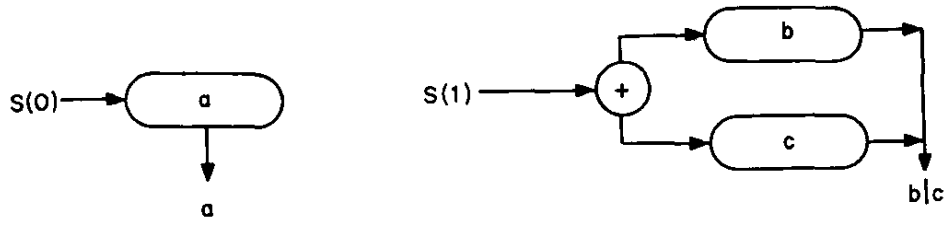


Figure 4.9: The next operator is an alternation operator `|`. This operator works with two topmost operands on the stack, in this example, with `b` and `c`. The result is a `CNODE b|c`. `CNODE` is represented by the plus sign in a circle. (taken from [46]).

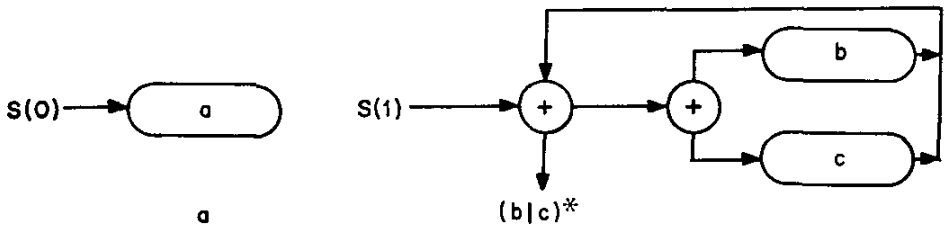


Figure 4.10: The next operator is the star operator. The star operator works only with a single topmost stack entry. In this example, it works with `CNODE b|c`. Same as for the alternation operator, a `CNODE` is used to realize the start operator. It is realized as follows: $(b|c)^* = \epsilon|(b|c)(b|c)^*$ (taken from [46]).

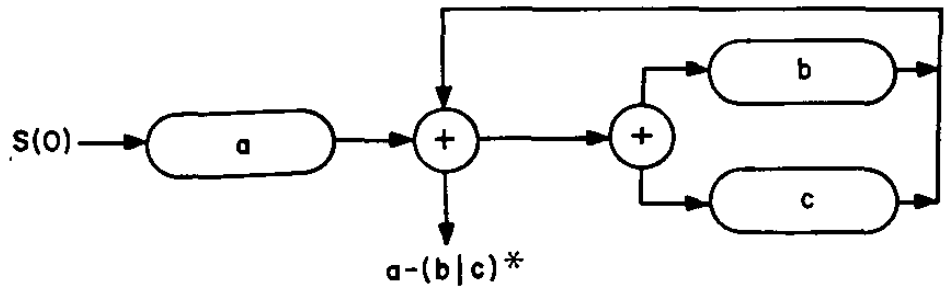


Figure 4.11: The next operator is the concatenation operator. It takes the two topmost stack entries and combines them to execute sequentially. In this example, it is the concatenation of the two segments created before. After this step, there will be only one operand on the stack, and that will be `a·(b|c)*` (taken from [46]).

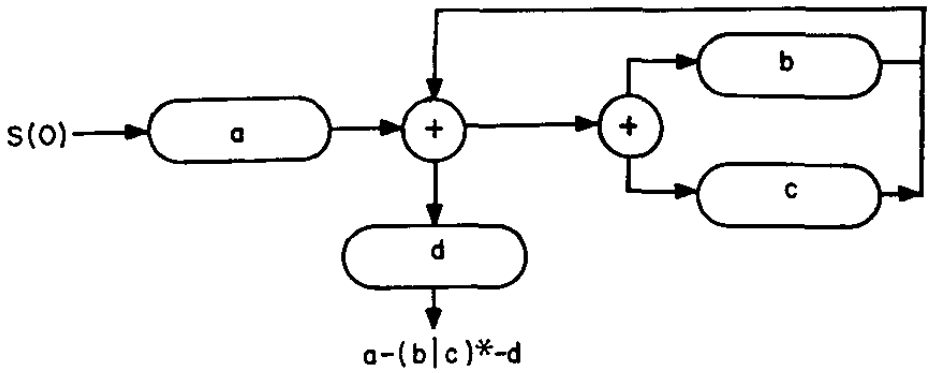


Figure 4.12: The final step consists of creating an NNODE from character `d` and then concatenating this operand with the only operand on the stack (regular expression `a·(b|c)*` from the previous step). This step will produce the final and only stack entry, which will be the regular expression `a(b|c)*d` itself (taken from [46]).

Thompson's Algorithm With Cache

Execution of a DFA is more efficient than the execution of an NFA. It is because the DFA will never have a multiple choice of the next states (i.e., it is only in one state at a time). A DFA can be created from any NFA. In such DFA, every state corresponds to a list of states of the NFA in which it can be in a given step [38]. An example of an NFA for the regular expression `abab|abbb` and a corresponding DFA is in Figure 4.13.

Even though the original paper by Thompson [46] talked about a list of next characters, it will be further called a list of next states to be consistent with the terminology of NFAs. In a sense, Thompson's algorithm computes a DFA state in each step. In a given step, it computes a list of the next states for a given character. The list of next states is the new DFA state. In that way, Thompson's NFA simulation is executing the equivalent DFA. After the state is used (i.e., the current list of states is processed, and the list of next states becomes the new current list), it is forgotten. Such a state has to be reconstructed when it is needed again. Rather than throw away the computed state after each step, it could be cached in spare memory. The caching will avoid the cost of repeated computing in the future. This approach essentially computes the equivalent DFA as is needed. NFAs derived

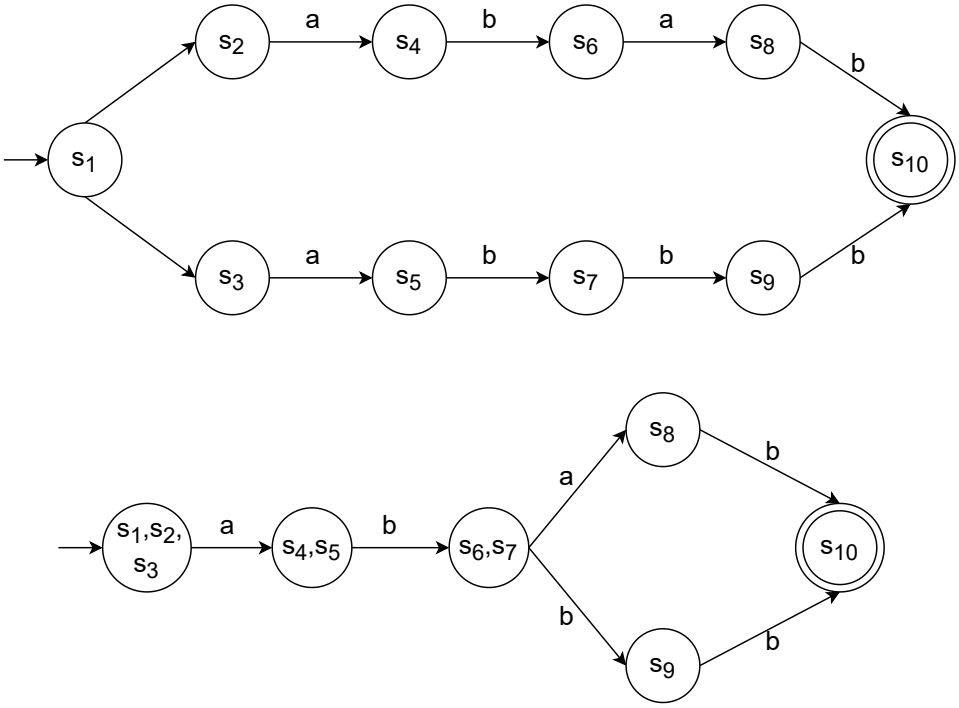


Figure 4.13: The first automaton is the NFA for the regex `abab|abbb`, the second automaton is the corresponding DFA. Each DFA state corresponds to a set of states, in which the NFA can be in a time. It can be in states s_1, s_2, s_3 as there are epsilon transitions from the state s_1 to states s_2 and s_3. From these states, the NFA can go to states s_4 and s_5 with the character `a`. Therefore, the next DFA state will be s_4, s_5 and there will be a new transition from the state s_1, s_2, s_3 to s_4, s_5 labeled with the character `a`. The next state s_6, s_7 is created analogically. Then, because there is a transition from the state s_6 to the state s_8 labeled with the character `a`, and a transition from the state s_7 to the state s_9 labeled with the character `b` in the NFA, there will be the same two transitions in the DFA. These two transitions can not be joined as they are labeled with different characters. The last two transitions from states s_8 and s_9 to the state s_{10} are copied from the NFA as these are the only choices for the states s_8 and s_9 (taken from [38] and edited).

from regular expressions tend to visit the same states and the same transitions when run on most texts. This makes the caching worth it; the first time a state is explored, it must be computed as in the NFA simulation. However, all the future explorations are just single memory access [38].

The aforementioned simple implementation of Thompson's algorithm by Cox [38] can be extended to use the caching by under a hundred lines of code. More details about the changes that need to be made in order to use caching and also about the implementation as a whole can be found in the same article by Cox [38].

Chapter 5

State-of-the-Art Matchers

This Chapter introduces the state-of-the-art tools used for regular expression matching. It provides an overview of each of the matchers, their algorithms, and optimization techniques.

5.1 Grep

Grep is a pattern matching engine. It searches the given patterns in the given files. It uses the above-mentioned Boyer-Moore algorithm for matching a single fixed pattern (i.e., performing the exact string matching). It also uses the Aho–Corasick algorithm, introduced in [5], for matching multiple fixed patterns [1].

Grep uses two algorithms for regular expression matching. The first algorithm is automata-based. More specifically, it is an optimized version of Thompson's on-the-fly determinization algorithm. The automata-based algorithm is used for as many regexes as possible since it is the faster option for regular expression matching. However, grep also supports backreferences in regexes. In general, the backreferences can not be implemented via the finite-state automaton. Therefore, it uses the backtracking algorithm mentioned above. The performance of the grep can be significantly worse when it uses the backtracking algorithm [1].

It also uses optimizations for both fixed pattern matching and regular expression matching. It uses raw system calls to get unbuffered input. It also looks for newlines only when the match is found to get the line with the match. In the Boyer-Moore algorithm, it unrolls the inner loop and sets up the delta table entries, so it does not need to do the loop exit test at every unrolled step. For the regular expression matching, it tries to extract a fixed string. If there is such a string, it must occur in every match. Grep tries to find the string in the input text using the Boyer-Moore algorithm. Then it checks the neighborhoods of the fixed string match with Thompson's algorithm to find the complete regex match [28, 35].

5.2 RE2

RE2 is a `C++` library, which provides an alternative to backtracking-based matching engines. Its primary goal is to provide a matching linear to the length of the input text. However, the linear-time constant may vary depending on the overhead of safe handling of the regular expression. It can be outperformed by backtracking-based matchers in various situations. It is because the RE2 acts pessimistically, whereas backtracking engines act optimistically. It

also does not implements all features of regexes, specifically those that require backtracking solution, such as backreferences [51].

RE2 uses optimization already when parsing the input regex. Users can use regexes that are not written efficiently. RE2 rewrites such a regex to its most efficient form. For example, singleton classes are used in order to avoid escaping the character (i.e., `[.]` instead of `\.`). Such singleton classes are rewritten to a single character. Another example is the alternation; for example, `a|b|c|d` can be efficiently expressed as a character class `[a-d]`). The rest of the algorithm then works with the simplified regex created during the parsing [39].

As the next step, RE2 compiles the regex to the NFA. The compiler compiles UTF-8 character classes down to an automaton that reads the input one byte at a time, so the UTF-8 decoding is built into the automaton. The output of the compiler is an instruction graph. The matching itself then uses an optimized version of Thompson's on-the-fly determinization algorithm. RE2 treats the DFA states as a cache. When the cache fills, it frees all the states and starts over again. Thanks to that, it is able to work in a fixed amount of memory [39].

The simple implementation described in [38] used a simple sequence field to do list insertion with duplicate elimination in constant time. RE2 does not store the state in the compiled program. However, the list insertion with duplicate elimination still should be implemented in constant time. RE2 uses a data structure named sparse set to accomplish that [39].

RE2 also uses various optimizations. It checks if the regex is matched in the input string but does not check where the match is, so it can look for the first literal byte. This optimization is done when every possible match starts with the same first byte (like in the regular expression `re2|random`). In such a case, the start of the match is found using `memchr`, which is faster than the general DFA loop. The DFA matching then starts from that position. In this type of match, where the position of the match is not important, the matching can also bail out early. For example, when searching for the regex `a+` in the text `ccaaaaaaaaaaaabd`, it can stop when it matches the first `a` in the text. This optimization is done by checking the match after every byte [39].

When it is also important to find where the match is in the input text, the DFA states are treated as a partially ordered set of NFA states instead of unordered sets. Then it prefers states for which the match starts earlier. Each time a match is found, it continues using only the states with equal or higher priority. Using this technique, it finds the end of the leftmost longest match. However, it also has to find the start of the match. It runs the DFA backward from the previously found end of the match. In this case, it treats all states equally and finds the longest possible match. The end of the longest backward match is the beginning of the original match [39].

If the goal is also to find sub-matches of the match, it uses a combination of the DFA matching to find the match and its boundaries. It then uses a direct NFA simulation to find the sub-matches in the already found match. The optimizations for this type of matching are more deeply discussed in [39].

5.3 Hyperscan

All the following information about the Hyperscan is adopted from [50], where the algorithms and optimizations are also discussed in more detail.

Hyperscan is a high-performance regular expression matcher with an own API written in C. It focuses mainly on network security applications on commodity server machines. It uses two core techniques for efficient pattern matching. It translates regular expression matching into series of string and finite automata matching using graph decomposition. In the Hyperscans novel approach, the string matching becomes part of the regular expression matching. This approach avoids wasting CPU cycles on duplicate matching. The matching DFA also tends to be smaller thanks to the decomposed regular expression, which increases the chance of fast matching. The second technique is an acceleration of both string and automata matching with SIMD operations.

The main idea behind the decomposition of the regular expression is that a disjoint set of string and sub-regex (or FA) components is created from the regex. Each of the components is then used for the match until the full match is found. The string components are a stream of literals. The regex components are then all the input regex parts that remain after the string components extraction. The regex components include one or more metacharacters that have to be translated into an FA for matching. The string matching is the first step, which finds all string components in the input text. Each of the found matches of the string can start a neighbor FA matching. Such an approach minimizes the waste of CPU cycles caused by unnecessary FA matching since the FA matching is executed only when needed.

The second part of the Hyperscan is the multi-string and FA matching that takes advantage of SIMD operations of the modern CPUs. The multi-string matcher is called *FDR*. The purpose of the FDR is to find candidate input strings that are likely to match some string pattern and verifies them to confirm the exact match. It performs extended *shift-or matching* to accomplish that. The successful string match often triggers the FA component matching. The FA component matching uses the state-of-the-art DFA matching or the NFA matching if the number of DFA states exceeds a threshold. For both of them, it takes advantage of the SIMD operations.

5.4 Symbolic Regex Matcher

Symbolic regex matcher (SRM) is a `.NET` matching tool. Its core matching algorithms are based on symbolic derivatives. Thanks to that, it supports extended regular expression operations such as intersection and complement. It supports the bounded loop quantifiers as well as a large set of common features. It also supports full `UTF16` encoded strings. Besides the matching, it also supports match generation [40].

SRM uses the same parser as the `.NET` regex engine. However, it uses a new backend engine, which is derivative-based. SRM works with derivatives of symbolic extended regular expressions. Extended refers to the allowed intersection, complement, and bounded qualifiers. Instead of using singleton classes as the basic building blocks of single character regexes, SRM uses predicates [40].

SRM provides backtracking free matching. Its complexity is linear in the length of the input text. It uses two key optimizations. First, it maintains the DFA in the form of an integer array where the indexes are regex nodes internalized into integers. Second, if it is applicable, it uses `string.IndexOf` to search the relevant initial prefix [40]. The evaluation in [40] shows that SRM outperforms `.NET` matcher on most of the regexes, and it offers performance comparable to the RE2.

Chapter 6

Counting-Set Automata for Regular Expression Matching

This Chapter focuses on counting-set automaton (CsA) and its usage in regular expression matching. The first step of using this automaton for pattern matching is to create it from the regular expression. Creating the CsA from the regex consists of translating the regex with counting into counting automata (CA), which is a non-deterministic automaton with bounded counters. Then the CA is determinized, and the output is CsA. When the CsA is created in such a way, its size does not depend on the repetition bounds used in the regex. In contrary to the DFA which size is exponential to the repetition bounds [48]. This Chapter is adopted from [48], where the CsA is introduced.

6.1 Counting Automata

Classical counter automata have counters that correspond to a counted sub-expression of a regex. Guards on transitions of the classical counter automata enforce a specified number of repetitions before the automata can move on, i.e., the counters are only supposed to count the number of passes of such parts. Counting automata (CAs) are a limited sub-class of classical counter automata for regexes with counting.

Definition 6.1.1. *A `counting algebra` is an effective Boolean algebra $\mathbb{C}$ associated with finite set C of counters.*

The counters have a *lower bound* $\boldsymbol{min}_c \geq 0$ and an *upper bound* $\boldsymbol{max}_c > 0$ such that $\boldsymbol{min}_c \leq \boldsymbol{max}_c$. These bounds correspond with the counted repetition bounds in the regex. The counters are used as a bounded loop variable.

Definition 6.1.2. *`Counter memories` are the set of interpretations $\mathfrak{m} \colon C \to \mathbb{N}$ such that $\forall c \in C \colon 0 \leq \mathfrak{m}(c) \leq \boldsymbol{max}_c$.*

Counter memories form the universe $\mathfrak{D}_{\mathbb{C}}$ of the effective Boolean algebra $\mathbb{C}$. The set of predicates $\Psi_{\mathbb{C}}$ of the algebra contains combinations of basic predicates $\textsc{canExit}_c$ and $\textsc{canIncr}_c$ for $c \in C$. The semantic of these predicates is defined as follows:

$$\mathfrak{m} \models \textsc{canExit}_c \iff \mathfrak{m}(c) \geq \boldsymbol{min}_c,$$
$$\mathfrak{m} \models \textsc{canIncr}_c \iff \mathfrak{m}(c) < \boldsymbol{max}_c$$

Definition 6.1.3. *`Counting automaton` is a tuple $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$, where:*

- $\mathbb{I}$ *is an effective Boolean algebra called the input algebra,*
- C *is a finite set of counters with an associated counter algebra* $\mathbb{C}$*,*
- Q *is a finite set of states,*
- $q_0 \in Q$ *is the initial state,*
- $F \colon Q \to \Psi_c$ *is the final state condition,*
- $\Delta \subseteq Q \times \Psi_{\mathbb{I}} \times (C \to \mathcal{O}) \times Q$ *is the (finite) transition relation, where* $\mathcal{O} = \{\textit{EXIT}, \textit{INCR}, \textit{EXIT1}, \textit{NOOP}\}$ *is the set of counter operations.*

The component f of a transition $(p, \alpha, f, q) \in \Delta$ is its (counter) operator. It is often viewed as the set of indexed operations $\textsc{op}_c$, where OP denotes the operation assigned to the counter c, $f(c) = \textsc{op}$.

To define the semantics of counter operators f each indexed operation $\textsc{op}_c$ is associated with a counter guard $\mathtt{grd}(\textsc{op}_c)$ and a counter update $\mathtt{upd}(\textsc{op})$, defined as follows:

$$\begin{aligned}
\mathtt{grd}(\textsc{noop}_c) &\stackrel{\text{def}}{=} \top_{\mathbb{C}} & \mathtt{upd}(\textsc{noop}) &\stackrel{\text{def}}{=} \boldsymbol{\lambda} n.n,\\
\mathtt{grd}(\textsc{incr}_c) &\stackrel{\text{def}}{=} \textsc{canIncr}_{\mathbb{C}} & \mathtt{upd}(\textsc{incr}) &\stackrel{\text{def}}{=} \boldsymbol{\lambda} n.n+1,\\
\mathtt{grd}(\textsc{exit}_c) &\stackrel{\text{def}}{=} \textsc{canExit}_{\mathbb{C}} & \mathtt{upd}(\textsc{exit}) &\stackrel{\text{def}}{=} \boldsymbol{\lambda} n.0,\\
\mathtt{grd}(\textsc{exit1}_c) &\stackrel{\text{def}}{=} \textsc{canExit}_{\mathbb{C}} & \mathtt{upd}(\textsc{exit1}) &\stackrel{\text{def}}{=} \boldsymbol{\lambda} n.1
\end{aligned}$$

The operation NOOP does not modify the value of the counter, and its guard is always true, i.e., it is always enabled. The guard of the operation INCR is enabled if the counter has not yet reached its upper bound. The operation INCR increments the counter. Guard of both EXIT and EXIT1 enables the corresponding operation when the counter reaches its lower bound. The operation EXIT resets the counter value to zero on exit from the counting loop. The operation EXIT1 is operation EXIT followed by INCR, i.e., the counter value is one after this operation.

A predicate $\varphi_f \in \Psi_{\mathbb{C}}$ over counter memories is the guard of a counter operator $f \colon C \to \mathcal{O}$. Update of the counter operator $\mathtt{f} \colon \mathfrak{D}_{\mathbb{C}} \cup \{\bot\} \to \mathfrak{D}_{\mathbb{C}} \cup \{\bot\}$ is a counter-memory transformer. The guard and the update are defined as follows:

$$\varphi_f \stackrel{\text{def}}{=} \bigwedge_{\textsc{op}_c \in f} \mathtt{grd}(\textsc{op}_c) \qquad \mathtt{f}(\mathfrak{m}) \stackrel{\text{def}}{=} \begin{cases} \boldsymbol{\lambda} c.\mathtt{upd}(f(c))(\mathfrak{m}(c)) & \text{if } \mathfrak{m} \models \varphi_f \\ \bot & \text{otherwise} \end{cases}$$

If $\mathfrak{m}$ satisfies the guard, $\mathtt{f}$ updates all counters in a counter-memory $\mathfrak{m}$ by their corresponding operation. When the guard is not satisfied, the result is $\bot$.

The *configuration automaton* $\mathrm{FA}(A)$ of CA A defines the language semantics of the CA A. Configurations of the CA A are pairs $(q,\ \mathfrak{m}) \in Q \times \mathfrak{D}_{\mathbb{C}}$ consisting of a state q and a counter-memory $\mathfrak{m}$. The $\mathrm{FA}(A)$ is defined as follows:

Definition 6.1.4. *A* ***configuration automaton*** *FA(A) of CA A is a symbolic finite automaton whose:*

- *states are the configurations of A (there are finitely many configurations of A),*
- *initial state is the initial configuration* $(q_0, \{c \mapsto 0 \mid c \in C\})$ *of A,*

- *state* $(p, \mathfrak{m})$ *of FA(A) is final iff* $\mathfrak{m} \models F(p)$,
- *a transition relation is defined as* $\Delta_{FA(A)} = \{((p, \mathfrak{m}), \alpha, (q, f(\mathfrak{m}))) \mid (p, \alpha, f, q) \in \Delta, \mathfrak{m} \models \varphi_f\}$.

If CA A is deterministic, then FA(A) is also deterministic. A is deterministic iff $\forall p \in Q$ and $\forall (p, \alpha_1, f_1, q_1), (p, \alpha_2, f_2, q_2) \in \Delta$: if both $\alpha_1 \wedge \alpha_2$ and $\varphi_{f_1} \wedge \varphi_{f_2}$ are satisfiable, then $q_1 = q_2$ and $f_1 = f_2$.

If for any two transitions (q, α, f, r) and (q', α', f', r'), either $\alpha = \alpha'$ or $[\![\alpha]\!] \cap [\![\alpha']\!] = \emptyset$, then A is simple. That means that different character guards do not overlap and can be mostly treated as plain symbols. The algorithm presented in Section 6.2 produces simple CAs. Example of an intuitive notation of CA, with the initial state q and final conditions $F(q) = \bot, F(s) = \text{EXIT}_c$, where $\boldsymbol{min}_c = \boldsymbol{max}_c = 100$ is in Figure 6.1. Figure 6.5 shows a CA in a more formal notation.

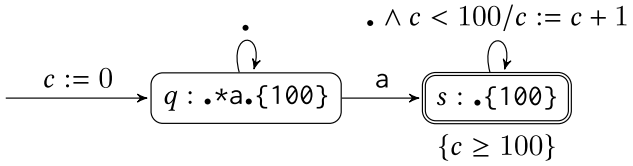


Figure 6.1: Example of the CA for the regex $.*a.\{100\}$ in an intuitive notation. The transitions are labeled by their guard, which gives the character class. On the left side of the „/“ delimiter can also be a guard of OP_c, which is shown in conjunction with the character guard α. On the right side of the delimiter, there is the update of OP_c written as an assignment to c. Specifically, the right side of the assignment for INCR_c is $c + 1$, for EXIT_c, it is 0, for EXIT1_c, it is 1, and NOOP_c is omitted. The transition does not change the value of the counter if it does not have any update specified. The initial state is labeled with the *initial value* of the counters. Final states are labeled with an *acceptance condition*. In this figure, it is the condition $\{c \geq 100\}$. Taken from [48].

6.2 Translating a Regex Into a CA via Conditional Partial Derivatives

A generalization of Antimirov's partial derivative construction [6] to *symbolic* counting is introduced in [48]. This generalization allows replacing a verbose NFA with succinct CA. The difference between the older variant of Antimirov's partial derivative construction [6] with *explicit* counting [40] and a version introduced in [48] will be illustrated in the example of the regex $.\{100\}$. From a hundred partial derivatives $\partial_.(.\{i\}) = .\{i - 1\}, 1 \leq i \leq 100$, and an NFA with a hundred states and transitions $(.\{i\}, ., .\{i - 1\})$ created by the older variant, to the single derivative $\partial_.(.\{100\}) = \{.\{100\}\}$ associated with a conditional counter update resulting in an NFA with a single state and the transition $(.\{100\}, \alpha, incr_c, .\{100\})$ created by the newer version from [48].

Before the construction takes place, regexes must be normalized by the following rules, where $x \rightsquigarrow y$ denotes that x is rewritten to y:

- The flattened right-associative *list form* must be maintained throughout the construction, i.e., all nested concat nodes must be rewritten using these rules: $(X \cdot Y) \cdot Z \rightsquigarrow X \cdot (Y \cdot Z)$, $\epsilon \cdot Z \rightsquigarrow Z$, and $Z \cdot \epsilon \rightsquigarrow Z$.

- The rule $S\{l,k\} \rightsquigarrow S\{0,k\}$ must be used for all S that are nullable. S can also be considered not nullable in the nullable context $S\{0,k\}$.

The size of the regex may decrease, or it remains the same after the normalization.

Let R be a fixed normalized regex. A *counting loop* is a sub-expression X of the regex R that is of the form $X = S\{l,k\}$. A *counter* is represented by a counting loop, and it is named after the loop itself. The *upper bound* of such counter is $\boldsymbol{max}_X = k$, and the *lower bound* is $\boldsymbol{min}_X = l$. An example can be the regex $(.\{9\})*$, which has the counter $X = .\{9\}$ whose bounds are $\boldsymbol{min}_X = \boldsymbol{max}_X = 9$. Further in the text, the set of all counters that occurs in R will be denoted by C (it can also be denoted by $\texttt{Counters}(R)$). For normalized regexes X and Y, the juxtaposition XY is again a normalized regex that is equivalent to the concat node $X \cdot Y$. For regexes $X = a\ \cdot b$, $Y = (a \cdot b)*$ these conventions means, that the juxtaposition $XY = a\ \cdot (b \cdot (a \cdot b)*$. I.e., concatenated elements are treated as sequences; the element itself is then a singleton sequence.

The construction works over the alphabet $\Sigma = Minterms(R)$, whose elements are minterms of R. Symbols from the alphabet Σ are used on the transition of the CA. The resulting CA created by this construction will also be simple.

Parametric Languages

In order to define the language of a normalized regex starting with a counting loop relative to a counter value, the definition of languages is lifted to be parametric in counter memories $\mathfrak{m}$. However, other regexes, i.e., regexes without counting loop, are treated the same as without the memory $\mathfrak{m}$, which is passed through unchanged.

For a counter operator f and a counter-memory $\mathfrak{m}$, $\mathtt{f}(\mathfrak{m})$ denotes appropriately updated memory. When f is not enabled, then $\mathtt{f}(\mathfrak{m}) = \bot$. Further in the text, if there is only a single counter $c \in C$ such that $f(c) \neq \textsc{noop}$, the counter operator f is sometimes identified with $\textsc{op}_c$. Also, $\textsc{op}_c(\mathfrak{m})$ can be used to represent the updated memory $\mathtt{f}(\mathfrak{m})$. Specifically, if enabled, $\textsc{incr}_X$ and $\textsc{exit}_X$ increments the value of the counter X by one and resets the value of the counter X to zero, respectively. The *parametric languages* of regexes are then defined as follows:

Definition 6.2.1. *Let $\mathfrak{m}$ be a counter-memory. Then the following equations define the parametric languages of regexes:*

$$L^{\mathfrak{m}}(\epsilon) \stackrel{def}{=} \{\epsilon\} \tag{6.1}$$

$$L^{\mathfrak{m}}(\psi Z) \stackrel{def}{=} [\![\psi]\!] \cdot L^{\mathfrak{m}}(Z) \tag{6.2}$$

$$L^{\mathfrak{m}}((W|Y)Z) \stackrel{def}{=} L^{\mathfrak{m}}(WZ) \cup L^{\mathfrak{m}}(YZ) \tag{6.3}$$

$$L^{\mathfrak{m}(S*Z)} \stackrel{def}{=} L^{\mathfrak{m}}(S)^* \cdot L^{\mathfrak{m}}(Z) \tag{6.4}$$

$$L^{\mathfrak{m}}(S\{l,k\}Z) \stackrel{def}{=} L^{\mathfrak{m}}(S) \cdot L^{INCR_{S\{l,k\}}\mathfrak{m}}(S\{l,k\}Z) \cup L^{EXIT_{S\{l,k\}}\mathfrak{m}}(Z) \tag{6.5}$$

$$L^{\bot}(X) \stackrel{def}{=} \emptyset \quad (\forall X) \tag{6.6}$$

The intuition behind Equation 6.4 is that all possibly present counters in S are inactive on the level of $S*$. Since, for $X = S\{l,k\}$ and $\mathfrak{m}' = \textsc{incr}_X(\mathfrak{m}), k - \mathfrak{m}'(X) < k - \mathfrak{m}(X)$ if $\mathfrak{m}(X) < k$, and $\mathfrak{m}' = \bot$ if $\mathfrak{m}(X) = k$, the Equation 6.5 is well-defined.

Theorem 6.2.1 proven in [47] relates $L^{\mathfrak{m}}(R)$ with the non-parametric definition of regular languages. The initial memory maps all counters to zero and is denoted by $0 \stackrel{\text{def}}{=} \boldsymbol{\lambda} c.0$.

Theorem 6.2.1. *Let R be a normalized regex. Then $L^0(R) = \mathcal{L}(R)$.*

Conditional Derivation

For the counter operator f and the normalized regex X, a *partial conditional derivative* is a pair $\langle f, X\rangle$. With a counter-memory $\mathfrak{m}$, the partial conditional derivative $\langle f, X\rangle$ defines the language $L^{\mathfrak{m}}(\langle f, X\rangle) \stackrel{\text{def}}{=} L^{f(\mathfrak{m})}(X)$. I.e., the counter-memory $\mathfrak{m}$ is first updated by applying the counter operator f, and the regex X is then evaluated within that updated memory. The language will be empty if the counter operator f is not enabled in $\mathfrak{m}$.

Conditional partial derivatives form a finite set called a *conditional derivative*. Given a counter-memory $\mathfrak{m}$, the language defined by a conditional derivative D is defined through languages of partial conditional derivatives in D. It is the union of such languages, e.g., $L^{\mathfrak{m}}(D) \stackrel{\text{def}}{=} \bigcup_{d\in D} L^{\mathfrak{m}}(d)$.

Before conditional derivatives of a given regex can be defined, the concept of a *sequential composition* of the conditional derivatives must be defined. The sequential composition of conditional derivatives D and E is defined as follows:

$$D \otimes E \stackrel{\text{def}}{=} \{\langle f; g, X \cdot Y\rangle \mid \langle f, X\rangle \in D, \langle g, Y\rangle \in E, f; g \neq \bot\} \tag{6.7}$$

The $f; g \neq \bot$ component of the definition is the composed counter operator, and it is obtained as $f; g(\mathfrak{m}) = g(f(\mathfrak{m}))$. The case when $f; g = \bot$ and some other special cases of sequential compositions are discussed later on.

Conditional derivatives of a normalized regex are defined by the following equations:

$$\partial_\alpha(\epsilon) \stackrel{\text{def}}{=} \emptyset \tag{6.8}$$

$$\partial_\alpha(\psi Z) \stackrel{\text{def}}{=} \begin{cases} \{\langle \boldsymbol{ID}, Z\rangle\} & \text{if } \alpha \wedge \psi \text{ is satisfiable} \\ \emptyset & \text{otherwise} \end{cases} \tag{6.9}$$

$$\partial_\alpha((W \mid Y)Z) \stackrel{\text{def}}{=} \partial_\alpha(WZ) \cup \partial_\alpha(YZ) \tag{6.10}$$

$$\partial_\alpha(S * Z) \stackrel{\text{def}}{=} \partial_\alpha(S) \otimes \{\langle \boldsymbol{ID}, S * Z\rangle\} \cup \partial_\alpha(Z) \tag{6.11}$$

$$\partial_\alpha(XZ) \stackrel{\text{def}}{=} \partial_\alpha(S) \otimes \{\langle \textsc{incr}_X, XZ\rangle\} \cup \{\langle \textsc{exit}_X, \epsilon\rangle\} \otimes \partial_\alpha(Z) \tag{6.12}$$

It is assumed, that concatenations $X \cdot Y$ are normalized to the flattened right-associative list form mentioned above, $\alpha \in \Sigma$, **ID** denotes the identity function $\boldsymbol{\lambda} x.x$, and a counting loop $S\{l, k\}$ is denoted by X.

The operation $\textsc{incr}_X$ gets composed with the $\textsc{noop}_X$ operation in $\partial_\alpha(S)\otimes\{\langle \textsc{incr}_X, XZ\rangle\}$ in Equation 6.12. This composition will yield $\textsc{incr}_X$ again. It is because $S\{l, k\}$ can not occur in S. The composition $\textsc{exit}_X; \textsc{incr}_X$ can occur in $\{\langle \textsc{exit}_X, \epsilon\rangle\} \otimes \partial_\alpha(Z)$ in Equation 6.12 when Z starts with X. The result of this composition will be the operation $\textsc{exit}1_X$ as $\textsc{incr}_X$ is trivially enabled when the counter value of X is zero. The last possible composition of individual operations in Equation 6.12 is $\textsc{exit}_X; \textsc{exit}_X$. This composition will be well-defined when $\boldsymbol{min}_X = 0$. It is because $\textsc{exit}_x$ is always enabled for $\boldsymbol{min}_X = 0$. The result of such composition is then $\textsc{exit}_X$. However, when $\boldsymbol{min}_X > 0$, then the composition $\textsc{exit}_X; \textsc{exit}_X$ is undefined and does not contribute anything to the composition. This is correct behavior because the counter value of X is reset to zero by the first $\textsc{exit}_X$, which results in the second $\textsc{exit}_X$ not being enabled since X is not nullable. Intuitively, the second occurrence of X must be iterated at least once before it can exit.

Below are two examples. The first, Example 6.2.1, shows the computation of conditional derivatives using Equation 6.8–6.12. The second, Example 6.2.2, with Figure 6.2, explains the use of some of the counter operations in the CA by describing them in the context of the partial-derivative-based construction.

Example 6.2.1. *Computation of conditional derivatives for regex* $R = .*a\{1,3\}a\{1,3\}a$. *The counting loop* $a\{1,3\}$ *will be denoted by* X. R *has two minterms* `a` *and* `[^a]`. *Because of the normal form assumption, the computation starts with* $\partial_\alpha(S * Z)$*:*

$$\begin{aligned}
\partial_a(R) &= \partial_a(.) \otimes \{\langle \boldsymbol{ID}, R\rangle\} \cup \partial_a(XXa) \\
&= \{\langle \boldsymbol{ID}, R\rangle, \langle \mathit{INCR}_X, XXa\rangle, \langle \mathit{EXIT}1_X, Xa\rangle\} \\
\partial_a(XXa) &= \partial_a(a) \otimes \{\langle \mathit{INCR}_X, XXa\rangle\} \cup \{\langle \mathit{EXIT}_X, \epsilon\rangle\} \otimes \partial_a(Xa) \\
&= \{\langle \mathit{INCR}_X, XXa\rangle\} \cup \{\langle \mathit{EXIT}_X, \epsilon\rangle\} \otimes \{\langle \mathit{INCR}_X, Xa\rangle, \langle \mathit{EXIT}_X, \epsilon\rangle\} \\
&= \{\langle \mathit{INCR}_X, XXa\rangle, \langle \mathit{EXIT}1_X, Xa\rangle\} \\
\partial_a(Xa) &= \partial_a(a) \otimes \{\langle \mathit{INCR}_X, Xa\rangle\} \cup \{\langle \mathit{EXIT}_X, \epsilon\rangle\} \otimes \partial_a(a) \\
&= \{\langle \mathit{INCR}_X, Xa\rangle, \langle \mathit{EXIT}_X, \epsilon\rangle\} \\
\partial_a(a) = \partial_a(.) &= \{\langle \boldsymbol{ID}, \epsilon\rangle\}
\end{aligned}$$

The composition $\mathit{EXIT}_X; \mathit{EXIT}_X$ *in* $\partial_a(XXa)$ *is undefined and therefore removed.*

$$\begin{aligned}
\partial_{[\hat{}a]}(R) &= \partial_{[\hat{}a]}(.) \otimes \{\langle \boldsymbol{ID}, R\rangle\} \cup \partial_{[\hat{}a]}(XXa) \\
&= \{\langle \boldsymbol{ID}, R\rangle\} \\
\partial_{[\hat{}a]}(XXa) &= \partial_{[\hat{}a]}(a) \otimes \{\langle \mathit{INCR}_X, XXa\rangle\} \cup \{\langle \mathit{EXIT}_X, \epsilon\rangle\} \otimes \partial_{[\hat{}a]}(Xa) \\
&= \emptyset \otimes \{\langle \mathit{INCR}_X, XXa\rangle\} \cup \{\langle \mathit{EXIT}_X, \epsilon\rangle\} \otimes \emptyset = \emptyset \\
\partial_{[\hat{}a]}(Xa) &= \partial_{[\hat{}a]}(a) \otimes \{\langle \mathit{INCR}_X, Xa\rangle\} \cup \{\langle \mathit{EXIT}_X, \epsilon\rangle\} \otimes \partial_{[\hat{}a]}(a) \\
&= \emptyset \otimes \{\langle \mathit{INCR}_X, Xa\rangle\} \cup \{\langle \mathit{EXIT}_X, \epsilon\rangle\} \otimes \emptyset = \emptyset \\
\partial_{[\hat{}a]}(.) &= \{\langle \boldsymbol{ID}, \epsilon\rangle\} \\
\partial_{[\hat{}a]}(a) &= \emptyset
\end{aligned}$$

The language defined by $\partial_a(Xa)$ *in a valid counter-memory* $\mathfrak{m}$ *is the union of the languages* $L^{\mathit{INCR}_X(\mathfrak{m})}(Xa)$ *and* $L^{\mathit{EXIT}_X(\mathfrak{m})}(\epsilon)$. *The first language corresponds to the case of iterating the loop* X *(if the counter value of* X *is below three). The second language corresponds to the case of exiting the loop (if the counter value of* X *is at least one) and accepting* $\{\epsilon\}$.

Example 6.2.2. *Consider the regex* $(.\{9\})^*$. *The CA for this regex is in Figure 6.2. The initial state is the regex itself, and its only partial derivative is* $.\{9\}(.\{9\})^*$. *In this partial derivative, the body of the counting loop is incremented once. The condition* $\mathit{canIncr}_c$ *must hold to do the incrementation. Since the automaton is in its initial state and no transition was yet taken, the value of counter* c *is zero, and therefore the* $\mathit{canIncr}_c$ *condition holds trivially.*

There are two partial derivatives for the state $.\{9\}(.\{9\})^*$, *both leading back to the same state. The first case is when* $\mathit{canIncr}_c$ *holds (i.e.,* $c < 9$*), the second case is when the counting loop is conditionally nullable and is exited under the condition* $\mathit{canExit}_c$ *(i.e.,* $c \geq 9$*). In the first case, the counter* c *is incremented (this is denoted by* $c < 9/c{+}{+}$ *in the figure). In the second case, the value of the counter* c *is reset to zero, and then it is incremented*

as a result of taking the partial derivative of $(.\{9\})*$. *As stated before, this composition yields the operation* EXIT1. *That means that the condition* CANEXIT_c *must hold, the condition* CANINCR_c *holds trivially since the counter is reset to zero a step before. The regex representing the initial state is nullable; therefore, the initial state is unconditionally final. The state* $.\{9\}(.\{9\})^*$ *is final iff* CANEXIT_c *holds (this is marked by "F:" in the figure).*

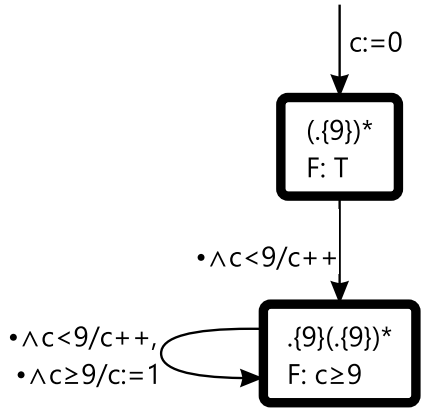


Figure 6.2: Counting automaton for the regex $(.\{9\})^*$ (taken from [48] and edited).

Theorem 6.2.2 states the correctness of the construction of conditional derivatives (the readers can refer to [47] for detailed proof). For this theorem, it is also necessary to define CANEXIT_R as the predicate shown above for a normalized regex R. Suppose that X stands for a counting loop, then CANEXIT_R is defined as follows:

$$\text{CANEXIT}_R = \begin{cases} \top_{\mathbb{C}} & \text{if } R = \epsilon, \\ \text{CANEXIT}_Z & \text{else if } R = YZ \text{ and } Y \text{ is nullable}, \\ \text{CANEXIT}_X \wedge \text{CANEXIT}_Z & \text{else if } R = XZ, \\ \bot_{\mathbb{C}} & \text{otherwise.} \end{cases} \tag{6.13}$$

Y may also be a counting loop in the second case in Equation 6.13. However, since it is stated that Y is nullable, its lower bound $\boldsymbol{min}_Y$ must be zero (due to the fact that R is normalized). Then CANEXIT_Y will always be true. Note that Z can also be ϵ in both the second and the third case. If $R = a\{0,3\}$ then the second case will be used with $Z = \epsilon$, which results in $\text{CANEXIT}_R = \top_{\mathbb{C}}$. If $R = a\{1,3\}$ then the third case will be used with $Z = \epsilon$, which results in $\text{CANEXIT}_R = \text{CANEXIT}_{a\{1,3\}} \wedge \top_{\mathbb{C}}$, i.e., CANEXIT_R is true if the loop $a\{1,3\}$ can be exited.

The following notions also have to be defined as they will be needed further in the determinization of CAs. A counter X is *visible in* R in these two cases:

1. $R = YZ$ and $X = Y$,
2. X does not occur in Y and X is visible in Z.

A counter-memory $\mathfrak{m}$ is *valid for* R if $\mathfrak{m}(X) = 0$ for all visible counters X that occur in R.

Theorem 6.2.2. *Let* R *be a normalized regex and let* $\Sigma = \mathit{Minterms}(\Theta)$ *where* Θ *is some finite superset of* $\mathit{Preds}(R)$. *If* $\mathfrak{m}$ *is valid for* R, *then* $L^{\mathfrak{m}}(R) = \bigcup_{\alpha \in \Sigma} [\![\alpha]\!] \cdot L^{\mathfrak{m}}(\partial_\alpha(R)) \cup \{\epsilon \mid \mathfrak{m} \models \text{CANEXIT}_R\}$.

Constructing CAs From Conditional Derivatives

Let $CA(R)$ be the counting automaton created from the regex R. The initial state of this automaton is R. The smallest set containing the initial state R and all regexes obtained from conditional derivatives constructed from R by repeated derivation wrt Σ is the set of states of the $CA(R)$. Let S be the regex that represents a state of $CA(R)$. There is a transition (S, α, f, T) in $CA(R)$ for each $\alpha \in \Sigma$ and each partial conditional derivative $\langle f, T\rangle \in \partial_\alpha(S)$. The *final condition* $F(S)$ of the state S is $\textsc{canExit}_S$. When S is not nullable and has no visible counters, then $F(S) = \bot_{\mathbb{C}}$, this corresponds to the classical case.

The paper by Turoňová et al. [47] shows that the following result can be proven by Theorem 6.2.2.

Theorem 6.2.3. *Let R be a normalized regex and $A = FA(CA(R))$. Then, for all $\langle \mathfrak{m}, S\rangle \in Q_A$, $\mathcal{L}_A(\langle \mathfrak{m}, S\rangle) = L^{\mathfrak{m}}(S)$.*

The construction of $CA(R)$ terminates, and the number of states of $CA(R)$ is linear in $\#_\Psi(R)$.

Theorem 6.2.4. *Let R be a normalized regex. Then $|Q_{CA(R)}| \leq \#_\Psi(R) + 1$.*

Proof of Theorem 6.2.4 can be found in Turoňová et al. [47]. The following final correctness result is a corollary of Theorem 6.2.1, Theorem 6.2.3, and Theorem 6.2.4.

Corollary 6.2.1. *Let R be a normalized regex. Then $\mathcal{L}(R) = \mathcal{L}(CA(R))$.*

Proof. *First, $Q_{CA(R)}$ is finite and thus well-defined by using Theorem 6.2.4. Use Theorem 6.2.3 with $\langle \mathfrak{m}, S\rangle$ as the initial state $\langle 0, R\rangle$ of A. It follows that $\mathcal{L}(A) = L^0(R)$. Then use Theorem 6.2.1 for $L^0(R) = \mathcal{L}(R)$ and $\mathcal{L}(CA(R)) = \mathcal{L}(A)$ holds by definition.* □

The number of input minterms of $CA(R)$ may be exponential in the number of predicates of R. However, when the predicates are represented as a finite union of intervals (which is typical for character classes), the size of a single predicate representation can be estimated to be proportional to the number of interval borders in the union. Since the total number of interval borders will remain the same in minterms as in the original set of predicates also the size of all minterms remains linear in the total size of all the predicates. That means that the mintermization based on character classes does not blow up the number of transitions in $CA(R)$. This was also experimentally validated in Turoňová et al. [48].

6.3 Determinization of Counting Automata

Counting automata created from the conditional derivative (as shown in Section 6.2) are non-deterministic. One approach for the determinization of *CAs* to *DFAs* is naive determinization. The naive determinization first converts the given *CA* to underlying *NFA*. This is done by making the counter memories an explicit part of control states. Then, the textbook subset construction is used to turn the *NFA* into the *DFA*.

The main disadvantage of this approach is the high risk of state explosion in one, or even worse, in both steps. The explosion is caused by two factors. In the first step, it is the sacrification of the succinctness of symbolic counters by making them part of the states. That makes the states linear in the counter bounds. In the second step, the explosion

is caused by the subset construction, which is exponential to the size of the NFA and, therefore, also to the counter bounds.

A new approach, introduced by Turoňová et al. [48], can handle the explosion problem. The new approach does a direct determinization of the CA into *counting-set automata (CsAs)*. $CsAs$ are a novel type of automata which control states produced by the direct determinization are essentially the states of the corresponding DFA without the counter memories. The states of CsA are equipped with special registers that can hold *sets* of integers in order to simulate the run of the DFA. These registers are used at runtime to compute the right values of the counters. This completely avoids the explosion caused by the wiring of counter memories into control states. The manipulation with a counting set can be implemented in constant time, making the simulation run fast.

Counting-Set Automata

This subsection introduces the formalized idea of counting-set automata. To allow manipulation with pairs of predicates from the input algebra $\mathbb{I}$ and the counting-set algebra $\mathbb{S}$, the notion of a combined Boolean algebra $\mathbb{I} \times \mathbb{S}$ is used. It is also assumed that predicates in $\Psi_{\mathbb{I}\times\mathbb{S}}$ have a form $\alpha \wedge \beta$ where $\alpha \in \Psi_{\mathbb{I}}$ and $\beta \in \Psi_{\mathbb{S}}$. The conjunction $(\alpha \wedge \beta) \wedge_{\mathbb{I}\times\mathbb{S}} (\alpha' \wedge \beta')$ has the usual meaning of $(\alpha \wedge_{\mathbb{I}} \alpha') \wedge (\beta \wedge_{\mathbb{S}} \beta')$ and $\alpha \wedge \beta$ is satisfiable if both α and β are satisfiable in their respective algebras.

The interpretation of counters is set-based, which means that the value of a counter c is a *finite set* rather than a single value. A counter is then called a *counting set*. Let $\mathcal{P}_{fin}(X)$ denote the powerset of X, which is restricted to finite non-empty sets. Then a function $\mathfrak{s}\colon C \to \mathcal{P}_{fin}(\mathbb{N})$, such that $\forall c \in C\colon Max(\mathfrak{s}(c)) \leq \boldsymbol{max}_c$ is called a *counting set memory* for C. Also, note that the set of all set memories for C is finite. An effective Boolean algebra $\mathbb{S}_C$ called the *counting-set algebra over* C is an algebra formed by counting-set predicates over C. When it is clear from the context, the counting-set algebra $\mathbb{S}_C$ is also denoted just by $\mathbb{S}$. The domain of the counting-set algebra $\mathfrak{D}_{\mathbb{S}}$ is the set of all set memories for C. The Boolean closure of the basic predicates canIncr_c and canExit_c forms the set of predicates $\Psi_{\mathbb{S}}$, which is syntactically the same as in counter algebra $\mathbb{C}$. However, since $\mathbb{S}$ is set-based, its semantic will differ from $\mathbb{C}$ to reflect this fact. The semantics of these predicates under counter algebra $\mathbb{S}$ is defined as follows:

$$\mathfrak{s} \models \text{canExit}_c \Leftrightarrow Max(\mathfrak{s}(c)) \geq \boldsymbol{min}_c,$$
$$\mathfrak{s} \models \text{canIncr}_c \Leftrightarrow Min(\mathfrak{s}(c)) < \boldsymbol{max}_c$$

where $Min()$ and $Max()$ denote functions that obtain the minimum and maximum of the set, respectively. The intuition behind these conditions is that it tests if at least one element of the set satisfies the counter condition.

Definition 6.3.1. *A counting-set automaton (CsA) is tuple* $A = (\mathbb{I}, C, Q, F, \Delta)$ *defined as follows:*

- $\mathbb{I}$ *is an effective Boolean algebra called the input algebra,*
- C *is a finite set of counters associated with the counting-set algebra* $\mathbb{S}$*,*
- Q *is the finite set of states,*
- $q_0 \in Q$ *is the initial state,*

- $F\colon Q \to \Psi_{\mathbb{C}}$ *is the final state condition,*
- $\Delta \subseteq Q \times \Psi_{\mathbb{I}\times\mathbb{S}} \times (C \to \mathcal{P}(\mathcal{O})) \times Q$ *is a finite set of transitions.*

The transition has four components, where the first and the last component is a state. The second component is the *guard* of the transition. The third component of the transition is the *counting-set operator*, where $\mathcal{O}$ denotes the set $\{\textsc{Incr}, \textsc{Noop}, \textsc{Rst}, \textsc{Rst1}\}$ of *counting-set operations*. The counting-set operations are basically counter operations, just lifted to sets. They are also written capitalized in order to distinguish them from the counter operations. Also, the lifted counter operations EXIT and EXIT1 are written as RST and RST1, respectively, to stress the different usage of these operations. The counting-set operator assigns sets of counting-set operations to counters; these sets are called *combined (counting-set) operations*.

Definition 6.3.2. *The CsA A is deterministic iff the following holds for every two transitions (p, ψ_1, f_1, q_1) and (p, ψ_2, f_2, q_2) in Δ: if $\psi_1 \wedge \psi_2$ is satisfiable, then $f_1 = f_2$ and $q_1 = q_2$.*

The semantics of an indexed counting-set operation $\textsc{op}_c \in \mathcal{O}$ is the set transformer $\mathtt{upd}(\textsc{op}_c)$, which is defined as follows:

$$\mathtt{upd}(\textsc{incr}_c) \stackrel{\text{def}}{=} \boldsymbol{\lambda} S.\{n+1 \mid n \in S \wedge n < \boldsymbol{max}_c\} \quad \mathtt{upd}(\textsc{rst}_c) \stackrel{\text{def}}{=} \boldsymbol{\lambda} S.\{0\},$$
$$\mathtt{upd}(\textsc{noop}_c) \stackrel{\text{def}}{=} \boldsymbol{\lambda} S.S \qquad\qquad \mathtt{upd}(\textsc{rst1}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} S.\{1\},$$

The counting-set operator $f\colon C \to \mathcal{P}(\mathcal{O})$ is assigned the counting-set-memory transformer $\mathtt{f}\colon \mathfrak{D}_{\mathbb{S}} \to \mathfrak{D}_{\mathbb{S}}$, which is defined as follows:

$$\mathtt{f} \stackrel{\text{def}}{=} \boldsymbol{\lambda} c.\begin{cases} \bigcup_{\textsc{op}\in f(c)} \mathtt{upd}(\textsc{op}_c)(\mathfrak{s}(c)) & \text{if } f(c) \neq \emptyset \\ \{0\} & \text{if } f(c) = \emptyset \end{cases} \tag{6.14}$$

Intuitively, if $f(c) \neq \emptyset$, there are some operations in $f(c)$ that have to be applied. The operations are applied on the value $\mathfrak{s}(c)$ of each counting set c, yielding counting sets for each $\mathfrak{s}(c)$. The new value of each $\mathfrak{s}(c)$ is then a union of its resulting counting sets. In the second case, when $f(c) = \emptyset$, there are no operations to apply. In this case, an implicit reset of c to $\{0\}$ (an implicit RST operation) is done. Such transitions are created by the determinization introduced in the Generalized Subset Construction when c is a dead variable (its value is irrelevant).

In terms of the guards, it is necessary to distinguish cases such as $\neg\textsc{canExit}_c \wedge \textsc{canIncr}_c$, $\textsc{canExit}_c \wedge \neg\textsc{canIncr}_c$, or $\textsc{canExit}_c \wedge \textsc{canIncr}_c$. Therefore, the *CsA* transition obtained throughout the determinization need guards that are partially independent of the operations of f. That is the reason why, as opposed to counter operators of a *CA*, a counting set operator f of *CsA* does not induce any guard, and the guard is instead an explicit part of the transition.

Also, there is a difference in updates in *CAs* and *CsAs*. That is because the semantic of the INCR operation is defined in such a way that it can not produce values greater than $\boldsymbol{max}_c$. The updates then have to be defined for *indexed* operations.

An underlying *configuration FA* of the *CsA* A, $FA(A)$, defines the *language of* A as $\mathcal{L}(A) := \mathcal{L}(FA(A))$. The individual components of the $FA(A)$ are then defined as follows:

- *Configurations* of A, i.e., tuples $(q, \mathfrak{s}) \in Q \times \mathfrak{D}_{\mathbb{S}}$ consisting of a state q and a counting-set memory $\mathfrak{s}$ forms a finite set of states of $FA(A)$,
- the *initial configuration* $(q_0, \{c \mapsto \{0\}\}_{c \in C})$ of A is the initial state of $FA(A)$,
- a transition $\tau = (p, \alpha \wedge \beta, f, q) \in \Delta$ is *enabled* in configuration $(p, \mathfrak{s})$ iff α is satisfiable and $\mathfrak{s} \in [\![\beta]\!]_{\mathbb{S}}$, i.e., $\mathfrak{s}$ satisfies the counter guard β; if τ is enabled in $(p, \mathfrak{s})$, then $FA(A)$ contains the transition $((p, \mathfrak{s}), \alpha, (q, f(\mathfrak{s}))$,
- a state $(q, \mathfrak{s})$ of $FA(A)$ is *final* iff $\mathfrak{s} \models F(q)$.

An example of CsA with an intuitive notation (also introduced in Section 6.1) is in Figure 6.3.

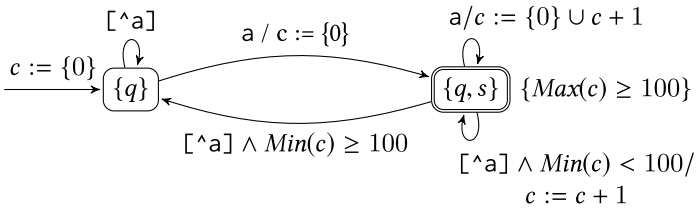


Figure 6.3: Example of the CsA for the regex $.*a.\{100\}$ in an intuitive notation, created by determinization of CA from Figure 6.1. The assignments to c are used to denote counting-set operators. Specifically, RST is written as assignation of $\{0\}$ to c, RST1 is analogical only with $\{1\}$ instead of $\{0\}$, INCR is represented by $c{+}1$, and NOOP is omitted. Also, transitions between the same states that differ only in guards are merged into one with a simplified guard. Taken from [48].

The main reason why the resulting machine for determinization of CAs is CsA is the fact that pattern matching with $CsAs$ is fast. Using appropriate data structure, all basic counting-set tests and updates, i.e., CANINCR$_c$, CANEXIT$_c$, NOOP, INCR, RST, and RST1, can be implemented to run in constant time (assuming constant-time complexity of integer arithmetics operations). The size of the counting set and the value $\boldsymbol{max}_c$ do not affect the complexity.

Combined counting-set operations can also be implemented to run in constant time. Although the union of two general sets could take linear time to the size of the sets (which is at most $\boldsymbol{max}_c$), the union of sets, where at most one is different from $\{0\}$ and $\{1\}$, can be computed in constant time. Only the operations NOOP and INCR may return sets other than $\{0\}$ and $\{1\}$. A *slow* transition is then a transition whose counting-set operator f assigns to some counter c the result of combined operation $f(c)$ that contains both NOOP and INCR. The CsA with the absence of *slow* transitions is then called *fast*; otherwise, it is called *slow*. According to Turoňová et al. [48], slow $CsAs$ are rare in practice.

In the used data structure, the *runtime value* of c is a tuple (o, l) where $o \in \mathbb{N}$ is called an *offset* and l is a queue of strictly increasing natural numbers such that $S_c = \{o - n \mid n \in l\}$. If the first and the last element of the queue can be accessed in constant time (such as a doubly-linked list), the data structure then supports the *constant-time* implementation of the following operations:

- the minimum and the maximum of S_c are obtained as $o - last(l)$ and $o - first(l)$, respectively,

- insert 0: if $o - last(l) > 0$, then append o at the end of l (similarly for inserting 1),
- incrementing all elements, up to $\boldsymbol{max}_c$: $o := o + 1$; if $o - first(l) > \boldsymbol{max}_c$, then remove $first(l)$,
- reset to $\{0\}$: $l := 0; o := 0$ (similarly for reset to $\{1\}$).

An example of usage of this data structure is in Figure 6.4.

prefix	state	(o, ℓ)	S_c
ϵ	$\{q\}$	$(0, [0])$	$\{0\}$
a	$\{q, s\}$	$(0, [0])$	$\{0\}$
aa	$\{q, s\}$	$(1, [0, 1])$	$\{1, 0\}$
aa0$^{(10)}$	$\{q, s\}$	$(11, [0, 1])$	$\{11, 10\}$
aa0$^{(10)}$aa	$\{q, s\}$	$(13, [0, 1, 12, 13])$	$\{13, 12, 1, 0\}$
aa0$^{(10)}$aab$^{(87)}$	$\{q, s\}$	$(100, [0, 1, 12, 13])$	$\{100, 99, 88, 87\}$
aa0$^{(10)}$aab$^{(87)}$d	$\{q, s\}$	$(101, [1, 12, 13])$	$\{100, 89, 88\}$
aa0$^{(10)}$aab$^{(87)}$df	$\{q, s\}$	$(102, [12, 13])$	$\{90, 89\}$
aa0$^{(10)}$aab$^{(87)}$dfa	$\{q, s\}$	$(103, [12, 13, 103])$	$\{91, 90, 0\}$

Figure 6.4: Example of the data structure during a run of CsA in Figure 6.3 over an input word $aa0^{(10)}aab^{(87)}dfa$. It shows the current state, the runtime counting-set configuration (o, l), and the value S_c that the (o, l) represents after processing of the prefix. The final condition of $\{q, s\}$ is fulfilled after processing the sixth and seventh prefix since the maximum of S_c is at least a hundred. Taken from [48].

Using the above-described data structure together with fast CsA for pattern matching, the tests and updates of one counting set take $\mathcal{O}(1)$ time. This results in the overall complexity of $\mathcal{O}(|C|)$ for all counting sets and their unions.

Encoding DFA Powerstates as CsA Configurations

This section describes two approaches that can be used for the configurations of a CsA to encode states of a DFA corresponding to NFA $FA(A)$ underlying a given CA $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$. Assume that the states of $FA(A)$ are pairs $(p, \mathfrak{m})$, where p is a state of A and $\mathfrak{m}$ is a counter-memory. The $FA(A)$ have such states due to the fact that A is converted to $FA(A)$ by making the counter memories explicit parts of states. Also, assume that the $FA(A)$ is determinized by the textbook subset construction. Considering a *simple* FA $\mathcal{A} = (\mathbb{I}, Q, q_o, F, \Delta)$, the set of states of the DFA created by the textbook subset construction will be $\mathcal{P}(Q)$, its transitions will be $(S, a, \{r \in Q \mid s(a)r \in \Delta, s \in S\})$, its initial state will be $\{q_0\}$, and its final states will be all those intersecting F. Explicit generation of all minterms in order to determinize CA that is not simple can be avoided by using a more sophisticated version of the subset construction for symbolic automata. This version of the subset construction is introduced in Veanes et al. [49]. Further in the text, the result of the textbook subset construction will be written as $DFA(A)$. Sets of states (i.e., sets of pairs $(p, \mathfrak{m})$) of $FA(A)$ then form the states of $DFA(A)$, called the *powerstates*. The CA-to-CsA determinization introduced in the Generalized Subset Construction builds CsA A' which control states are the subset of the set Q of the states of the CA A. Pairs $(R, \mathfrak{s})$ where $R \subseteq Q$ is a CsA control state, i.e., a set of states of A, and $\mathfrak{s}\colon C \to \mathcal{P}_{fin}(\mathbb{N})$ is a counting-set memory, are the configurations of A'.

The first approach to encode a powerstate to a CsA configuration is to interpret the configuration $(R, \mathfrak{s})$ as a DFA state containing all pairs (r, m) such that $r \in R$ and $\mathfrak{m}(c)$ can be any value from $\mathfrak{s}(c)$ for all $c \in C$. The set of the counter memories $\mathfrak{m}$ is then isomorphic to the Cartesian product $\Pi_{c \in C}\mathfrak{s}(c)$ of the sets $\mathfrak{s}(c)$ assigned to the counters, and the entire powerstate is the Cartesian product $R \times \mathfrak{m}$ of the set of states and the set of counter memories. This approach is called *naive encoding*, and since it can not express any dependence of a counter-memory on the CA state (every state can be paired with each considered memory), it is too impractical. It also can not express the mutual dependence of values of different counters within a counter-memory (every possible value of a counter c can be paired with every possible value of any other counter d). This encoding can not represent most of the $DFAs$ that arises from real-life regexes. An example can be the CA from Figure 6.1 and its DFA configuration $\{(q, c = 0), (s, c = 0), (s, c = 1)\}$, which can not be represented by the naive interpretation of a CsA configuration, since q and s appear with different sets of counter values.

The second approach is *encoding with counter scopes*. The major difference in comparison to naive encoding is considering the fact that the value of a counter is usually implicitly zero at most states. In other words, not every counter is used at every state of the CA. In such states (which must be known in order to use this approach), the implicit zeros can be omitted from the counting sets, making the encoding much more flexible. To formalize this concept, the paper by Turoňová et al. [48] introduces the notion of the *scope of a counter* that over-approximates the set of states where a counter c can have a non-zero value. In a general case, computing a precise set of such states would require a reachability analysis. That is because some of the transitions may never be used. For example, when simultaneously counting with two counters c and d for which $\textsc{canIncr}_c < \textsc{canExit}_d$, the exit transition for d can not be used since the $\textsc{canExit}_d$ guard will never be satisfied. However, since the derivative construction produces CAs without such transitions, the scope corresponds to this set precisely, and therefore the set of states where a counter c can have a non-zero value is easy to compute. The scope is defined inductively as follows:

Definition 6.3.3. *The scope is the smallest set of states $\sigma(c)$ such that $q \in \sigma(c)$ if:*

- *there is a transition to q with either $\textsc{incr}_c$ or $\textsc{exit1}_c$,*
- *there is a transition to q from a state in $\sigma(c)$ with $\textsc{noop}_c$ operation.*

I.e., the state is in scope if the counter c gets incremented on the incoming transition and the scope then spreads along with the transition relation until a transition with $\textsc{exit}_c$ takes place.

The formal definition is then the following: *The DFA powerstate encoded by a CsA configuration $(R, \mathfrak{s})$ is the set $(R, \mathfrak{s})^{DFA}$ of configurations $(r, \mathfrak{m})$ of the CA A such that $r \in R$ and, for all $c \in C, \mathfrak{m}(c) \in \mathfrak{s}(c)$ if $c \in \sigma(r)$, else $\mathfrak{m}(c) = 0$.* The powerstate of $DFA(A)$ is called *Cartesian* if it can be encoded by CsA configuration. The $DFA(A)$ is then called Cartesian if all its powerstates are Cartesian.

For example, considering CsA A from Figure 6.1, since q_0 is not in the scope of c, the powerstates of the $DFA(A)$ are Cartesian.

Unfortunately, not all kinds of DFA powerstates can be expressed by the Cartesian encoding. Specifically, more subtle dependencies of counter values on the state and dependencies of counter values on each other can not be expressed by the Cartesian encoding. These dependencies mostly arise from regexes with nested counting sub-expressions, which

are compiled in CAs with nested counting loops. An example of a regex that is compiled to non-Cartesian CA is $(a|aa)\{5\}$. More information about the non-Cartesian powerstates can be found in the paper by Turoňová et al. [48], which also provides strong empirical evidence that a significant majority of real-life regexes lead to Cartesian CA.

Generalized Subset Construction

This section describes the core of the CA-to-CsA determinization, which is built on top of the textbook subset construction for $NFAs$. Since the derivative construction introduced in Section 6.2 generates simple CAs (their transitions are labeled with minterms of the original regex, and therefore different character classes on its transitions do not overlap), it is assumed that input CAs of the determinization are simple. However, it could be generalized to work with non-simple CAs in the style of symbolic automata determinization of Veanes et al. [49].

Let $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$ be a simple CA with the scope function $\sigma\colon Q \to \mathcal{P}(C)$. The deterministic CsA $A' = (\mathbb{I}, C, Q', S_0, F', \Delta')$ is then the result of the determinization algorithm. The components of A' are constructed as follows:

- $\mathbb{I}$ and C stays the same,
- $Q' \subseteq \mathcal{P}(Q)$ (the control states of A' are called powerstates),
- initial powerstate is $S_0 = \{q_0\}$,
- $F'(S) \stackrel{\text{def}}{=} \bigvee_{q \in S} F(Q)$, in other words, the powerstate $S \in Q'$ is final iff the final condition holds for any of its element,
- The sets Δ' and Q' are constructed by a fixpoint computation that explores the state space reachable from S_0. When a new transition from an already reached powerstate is created during the construction, it is added to Δ'. Also, the target powerstate of the newly created transition is added to Q'. When no new powerstate can be reached, the sets Δ' and Q' are complete.

Transitions of the CsA A' must be constructed in such a way that their updates of the runtime values of counting sets simulate transitions of the DFA corresponding to the CA A. Let $(R, \mathfrak{s})$ be a CsA configuration and $((R, \mathfrak{s})^{DFA}, \alpha, P)$ be a DFA transition from the DFA powerstate encoded by $(R, \mathfrak{s})$ over an input minterm α. A configuration $(R, \mathfrak{s})$ must be transformed into $(R', \mathfrak{s}')$ with $(R', \mathfrak{s}')^{DFA} = P$ by simulating the CsA transition. The CA $\alpha-$transitions enabled in configurations $(r, \mathfrak{m}) \in (R, \mathfrak{s})^{DFA}$ are instantiations of $\alpha-$transitions of the NFA $FA(A)$ from which the simulated DFA transition was constructed. These CA transitions will be used to construct the simulating CsA transition. Before the construction takes place, it is necessary to delete some CA transitions, which can never be taken. Such transitions, however, can create CsA transitions without corresponding guard because the source state is not in scope. These CsA transitions then cause that the CsA can accept a different language than the CA. It is a transition from the state, which is not in scope, with EXIT or EXIT1 operation for counter with lower bound greater than zero. Since the counter is not in the scope of the source state, its value will be zero, and such transition can never be taken since the CANEXIT guard can never be satisfied. Let Σ be the set of minterms over all input predicates in the CA A. The CA transitions can then be identified by the following:

1. source state, which must be in R,
2. an alphabet minterm $\alpha \in \Sigma$,
3. compatibility with a particular set of enabled or disabled counter guards.

The set of guards mentioned in case 3 above belongs to the set of minterms $\Gamma_{R,\alpha}$ of the set of counter guards on the $\alpha-$transitions originating in R, which is defined as follows:

$$\Gamma_{R,\alpha} \stackrel{\text{def}}{=} Minterms(\{\texttt{grd}(\text{OP}_c) \mid (r,\alpha,f,s) \in \Delta, r \in R \wedge c \in \sigma(r), \text{OP}_c \in f\}) \tag{6.15}$$

For each $\alpha \in \Sigma$ and $\beta \in \Gamma_{R,\alpha}$ there will be a transition leaving R in the CsA. The set of CA $\alpha-$transitions originating in R and *consistent* with β that is used to build that CsA transition is defined as follows:

$$\Delta_{R,\alpha,\beta} \stackrel{\text{def}}{=} \{(r,\alpha,f,s) \in \Delta \mid r \in R, \boldsymbol{Sat}(\varphi_f \wedge \beta)\} \tag{6.16}$$

All target states of the transitions in $\Delta_{R,\alpha,\beta}$ forms the set T which is the target powerstate of the newly created transition of CsA. The guard of this transition is $\alpha \wedge \beta$ (the predicates in $\Psi_{\mathbb{C}}$ and $\Psi_{\mathbb{S}}$ are syntactically the same).

As stated before, the transition must simulate the updates of transitions from which it is created. So, the last component of the transition, the counting-set operator f' must summarize the updates of the counter values on transitions of $\Delta_{R,\alpha,\beta}$ as updates of the respective counting sets. It must also take the scope of the counter into consideration. Tracking of the value of the counter starts when A' simulates a transition of A entering the scope of the counter. The tracking then ends when no state from the scope is present in the target CsA state. In all other situations, when the counter is out of scope, its value is implicitly zero, and the counter will not be tracked in counting sets.

Let $\Delta_{R,\alpha,\beta}(c)$ be the set of transitions from the set $\Delta_{R,\alpha,\beta}$ whose target state is in the scope of c. Also, let $op(\tau, c)$ denotes the counting-set operation that for a given CA transition $\tau = (p,\alpha,f,q)$ transforms the set of possible values of the counter c at the state p to the set of possible values obtained at the state q after taking the transition. It is defined as follows:

$$op((p,\alpha,f,q),c) \stackrel{\text{def}}{=} \begin{cases} \textsc{Noop} & \text{if } f(c) = \textsc{noop} \wedge p \in \sigma(c) \\ \textsc{Incr} & \text{if } f(c) = \textsc{incr} \wedge p \in \sigma(c) \\ \textsc{Rst} & \text{if } f(c) = \textsc{noop} \wedge p \notin \sigma(c) \\ \textsc{Rst1} & \text{if } f(c) = \textsc{incr} \wedge p \notin \sigma(c) \\ \textsc{Rst} & \text{if } f(c) = \textsc{exit} \\ \textsc{Rst1} & \text{if } f(c) = \textsc{exit1} \end{cases} \tag{6.17}$$

The set operation induced by the CA transition corresponds to the counter operation on the transition. When the CA transition comes from out of the scope, the counter can only have the value zero, which is the same as produced by EXIT (or eventually EXIT1 if the counter is immediately incremented). This corresponds to the third and fourth cases in Equation 6.17.

The counting set operator f' is then defined as $f'(c) \stackrel{\text{def}}{=} \{op(\tau, c) \mid \tau \in \Delta_{R,\alpha,\beta}\}$. It can also end up empty. That happens in a situation when the target powerstate is fully out of the scope of c, which is semantically corresponding to the implicit reset to $\{0\}$. The resulting CsA transition is therefore $(S, \alpha \wedge \beta, f', T)$. Since for any two distinct transitions (S,α_1,f_1,S_1) and (S,α_2,f_2,S_2), the condition $\alpha_1 \wedge \alpha_2$ is unsatisfiable by virtue of minterms, the CsA A' is deterministic.

Theorem 6.3.1. *For the CA A and the CsA A' above, we have $\mathcal{L}(A') \supseteq \mathcal{L}(A)$ and $|Q'| \leq 2^{|Q|}$*

An idea of proof of Theorem 6.3.1 can be found in Turoňová et al. [48].

Example 6.3.1. *The counting automaton for the regex $a\{1,3\}b\{5,9\}ab$ is shown in Figure 6.5. The counting-set automaton created by determinization of the counting automaton is shown in Figure 6.6. The determinization process will be demonstrated on the creation of transition from the state S_0 to the state S_1 for the letter b. There is only one transition in the counting automaton that will be considered, and it is the transition from the state q_0 to the state q_1. There are two guards on the transition; however, only the counter X is in the scope of the state q_0, and therefore the guard for the counter Y is irrelevant. The set $\Gamma_{S_0,b}$ will then contain the following two elements: $\textsc{canExit}_X$ and $\neg\textsc{canExit}_X$. Since there is $\textsc{canExit}_X$ on the original transition, the element $\neg\textsc{canExit}_X$ of the $\Gamma_{S_0,b}$ set can not be used as $\textsc{canExit}_X \wedge \neg\textsc{canExit}_X$ is not satisfiable. The last missing component of the transition is the operation. The $\textsc{exit}_X$ operation of the CA transition is irrelevant since X is not in the scope of the state q_1. The $\textsc{incr}_y$ operation will be transformed to the $RST1$ operation by using the fourth case of Equation 6.17. The final created transition is then the transition $(S_0, b \wedge \textsc{canExit}_X, \{RST1_Y\}, S_1)$.*

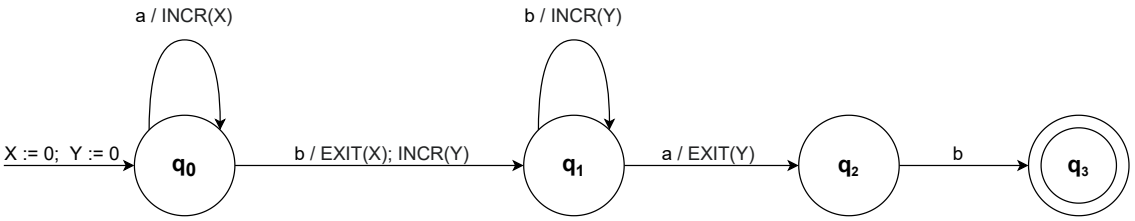


Figure 6.5: Counting automaton for the regex $a\{1,3\}b\{5,9\}ab$. X denotes the counter $a\{1,3\}$, and Y denotes the counter $b\{5,9\}$. Assignation of a value to the counter is denoted by := sign (e.g., X := 0). The character class (in this example, only a single letter) is written as first on the transition. If there are any operations, they are written after / symbol.

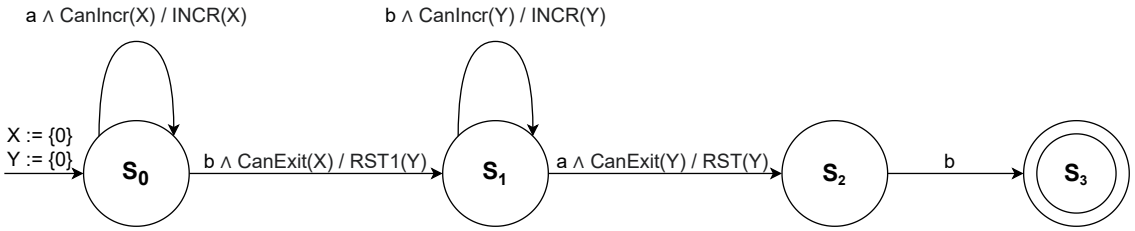


Figure 6.6: Counting-set automaton for the counting automaton from Figure 6.5. X denotes the counter $a\{1,3\}$, and Y denotes the counter $b\{5,9\}$. The character class (in this example, only a single letter) is written as first on the transition; then there are counter guards, and after / symbol, there are the operations. For both guards and operations, the counters on which they are applied are written as an argument.

Chapter 7

Implementation

This Chapter describes the implementation of the CsA-based pattern matching done within the RE2 matcher. It is further divided into individual sections following the steps to create CsA described in Chapter 6. The main steps are translating a regular expression into the CA, determinization of the CA, and the matching itself. Besides the implementation of the matching itself, there is a set of tests covering each of the mentioned steps and benchmarks for the first two parts of the algorithm (i.e., creating the counting automaton from the input regex and its determinization).

7.1 Translating a Regular Expression Into Counting Automaton

The first step of the algorithm is to create the CA from the input regex. This part of the algorithm works with already preprocessed regex along with some other information obtained through the processing. The output of this part of the algorithm is an instance of a `Regexp::Derivatives` class. The instance holds the resulting counting automaton together with some additional information needed further in the algorithm (for example, a set of all counters that occurs in the regex).

Preprocessing the Input Regex

The preprocessing of the input regex can be divided into two parts. The first part is done by already implemented functions of the RE2. This part loads the input regex in the form of a string. Then it will check the syntactical correctness of the regex and creates an instance of a `Regexp` class holding the regex itself. The `Regexp` class instance is an instance variable of an `RE2` class, which is the class that holds all information needed for the algorithm. The information that will be further used is the `use_CsAs_` and `unanchored_` options, the `bytemap_` array, and the `bytemap_range_`. Their usage will be described in more detail further in the text, but their meaning is briefly the following:

- `use_CsAs_` option is used to determine if the CsAs-based matching algorithms or the original algorithms should be used,
- `unanchored_` is used to determine if $.*$ should be prepended and appended to the regex, such modified regex is then used for partial matching,

- `bytemap_` is an array of size 256 where the index is an ordinal value of a corresponding character, and the value is the number of the bytemap class for the character,
- `bytemap_range_` is a number that determines the number of character classes.

The second part of the preprocessing, which is already part of the newly implemented algorithm, works with the output of the first part, specifically with the instance of the `Regexp` class. It normalizes the regex to the right-associative list form described in Section 6.2. It must also do some further modifications as part of the normalization. Specifically, it must transform regex operators for whose there is no equation defined in Equation 6.8–6.12 for partial derivative construction. The transformation is done so that there is no need to introduce new equations for the partial derivatives.

The operators are the *plus* operator (`+`) and the *quest* operator (`?`). The plus operator means one or more repetitions of the pattern. The only difference between the plus operator and the star operator is that there must be at least one repetition of the pattern for the plus operator. The plus operator is therefore transformed to the star operator with the repeated pattern as a prefix. For example, the regex `a+` will be transformed to the regex `aa*`. For the transformed regex, there must be at least one `a`, same as for the original regex with the plus operator, and then there can be an unlimited number of repetitions of `a`. Such transformation, therefore, creates an equivalent regex, which fits the equations for the partial derivatives construction.

The quest operator means zero or one occurrence of the pattern, e.g., the regex `aa?` will match two strings, `a` and `aa`. The operator will be transformed to *alternation* (i.e., `|` operator). One alternative of the transformed regex will be ϵ, which corresponds to the zero occurrences part of the quest operator. The second alternative will be the original pattern without the quest operator, which corresponds to the one occurrence part of the quest operator. For example, the regex `a?` will be transformed to the regex ϵ`|a`. The transformed regex match zero or one occurrence of the pattern, and therefore, it is equivalent to the original regex. Again, the transformed regex fit the equations for the partial derivatives construction.

Creating a Counting Automaton

After the regex is normalized using the steps above, it can be used to create the CA. It is created using Equation 6.8–6.12. They are implemented as several methods. The methods are the following:

`getEquationTypeAndOperands`

This method gets the type of equation that has to be used. It also gets the operands for the equation. The regex (i.e., the `Regexp` class instance) is passed as an argument to the method. The regex has an operator[1] associated with it and a list of subexpressions if there are any.

When the regex is just a single expression (i.e., it has no subexpressions), the concatenation equation will be used (Equation 6.9) for most of the cases. The first operand of the equation will be the expression itself, and the second will be ϵ. However, there are some special cases:

[1]The operators with comments can be found in the *regexp.h* file available at https://github.com/google/re2/blob/master/re2/regexp.h

- when the regex is of a type `kRegexpLiteralString` (for example, the regex `abcde`), the equation type will be the same, but the first operand will be `a`, and the second will be `bcde`,
- when the regex is of a type `kRegexpStar` (for example, the regex `a*`), the repetition equation will be used (Equation 6.11), and the operands stay the same,
- when the regex is of a type `kRegexpRepeat` (for example, the regex `a{100,300}`), the counted repetition equation will be used (Equation 6.12), and the operands stay the same.

When the regex is of a type `kRegexpAlternate`, it will be processed the same as the single expression regexes, even though it has some subexpressions. It is because the subexpression will be processed later in the computation. For such regexes, the alternation equation will be used (Equation 6.10). The first operand of the equation will be the expression itself, and the second will be ϵ.

When the regex is of a type `kRegexpConcat`, there are two ways of processing it. The first is when the type of its first subexpression is not `kRegexpConcat`. In such a case, it is processed similarly to the single expressions described above. The difference is that the type of the equation is determined based on the first subexpression. For example, for regexes `a{1,3}b{2,4}a` and `a*a`, the type of equation will be determined by `a{1,3}` and `a*`, respectively. The first operand will be the first subexpression of the regex. The second operand will be the regex without the first subexpression. For example, for the regex `a{1,3}b{2,4}a`, the first operand will be `a{1,3}` and the second operand will be `b{2,4}a`. There is also one special case. It is when the first subexpression is of the type `kRegexpLiteralString`. The second operand in such a case is not just the regex without the first subexpression. It is the concatenation of the first subexpression without the first literal and the rest of the regex. For example, for the regex `abca*`, the first operand will be `a`, and the second operand will be `bca*`.

The second way of processing the regex of the type `kRegexpConcat` is when its first subexpression has the `kRegexpConcat` type. Such a case can be caused by normalization. For example, the regex `(aa{1,3}b)a{1,4}` is rewritten to `aa{1,3}ba{1,4}` by the normalization. Then it is the concatenation of two expressions `aa{1,3}b` and `a{1,4}`. The first expression then has the `kRegexpConcat` type. The first subexpression must be processed as a standalone regex. The first operand is then set correctly. However, the second operand must be modified. It will be the concatenation of the second operand created by processing the first subexpression and the rest of the whole original regex.

`composition`

The method does the composition needed in Equation 6.11 and Equation 6.12. It creates new transitions that arise from the composition. The source state is the operand of the equation. The target state of the newly created transitions will be a new regex. The new regex is created as the concatenation of regexes from the partial derivatives that are arguments of the composition. The transitions will also have a new counter operator that arises from the composition. The new operator is computed by the `getOperatorComposition`, which must also consider the special cases described in the Conditional Derivation.

`computeNewState`

This is the main method of CA creation. It starts with the whole regex as it is the initial state, and then it explores every newly created regex (state). Even those states that will not be in the resulting CA because they will be unreachable. Their results are needed in other equations. The method first gets the equation type and its operands using the `getEquationTypeAndOperands` method. Then it implements Equation 6.8–6.12. All the equations must be computed for all bytemap classes.

For Equation 6.9, the method checks the satisfiability of the first operand. If it is satisfiable, it creates a new transition from the first operand to the second operand. The transition will also have the bytemap class associated with it. If the regex is of a type `kRegexpAnyChar`, the bytemap class is set to 256, which is beyond the upper bound of the bytemap. It is then used in the determinization to determine that the transition can be taken for any bytemap class. It also adds the second operand as a new state to be explored.

For Equation 6.10, the method first creates new regexes as the concatenation of individual subexpressions of the first operand with the second operand. For each of the new regexes, it computes the partial derivatives by calling itself (i.e., the `computeNewState` method). Then it copies all the transitions created by the newly created regexes and changes the source state to the equation operand in all of them.

For Equation 6.11, the method first computes partial derivatives for the pattern of the first operand (for example, for pattern `a` if the first operand is `a*`) and for the second operand. Then it computes the composition and also copies all of the transitions created by the second operand and changes the source state to the equation operand in all of them.

For Equation 6.12, the method first computes partial derivatives for the pattern of the first operand (for example, for pattern `a` if the first operand is `a{1,3}`) and computes the first composition defined in the equation. Then it computes partial derivatives for the second operand and computes the second composition defined in the equation. The result of this equation is the union of the two computed compositions. The last step is then to merge the transition that arises from the compositions together. The method also saves the counting loops while computing this equation. The counting loops are saved by their names (for example, the counting loop `a{1,3}` will be saved as `a{1,3}`). If there is more than one such counting loop, the others are saved with the number of already found counting loops with the same name appended to the end. For example, the regex `a{1,3}a{1,3}` will have two counting loops, `a{1,3}` and `a{1,3}1`.

In each of the equations, it is also determined if the regex (i.e., the operand of the equation) is a final state. The check is done using the `isNullable` method, which implements the CANEXIT_R predicated defined in Equation 6.13. The final state condition is `True` if the checked regex does not start with a counting loop or if it starts with a nullable counting loop. Otherwise, the final state condition will be CANEXIT of the corresponding counting loop.

Structure of the Resulting Counting Automaton

The counting automaton is saved in an `unordered_map` where the key is a string representing the source state, and the value is a vector of transitions. The transition is then a 5-tuple where the individual components are the following:

- a string representing the source state,
- an integer representing the bytemap class,

- a set of counter guards (the guards in the CA are not an explicit part of the transition; however, they are saved together with operations, so the operations do not have to be traversed later in the determinization to get the guards)
- a list of counter operations,
- a string representing the target state.

The class instance also contains some additional information about the counting automaton that will be used further in the determinization. It contains an `unordered_map` of all CA states. The key in this map is a string representation of the state, and the value is a pair representing the state. The pair contains an instance of the `Regexp` class representing the regex and a set of counters in scope (which will be computed during the determinization).

There is also an `unordered_map` of all final states. The key in this map is a string representation of the state. The value is a `finalStateCondition`, which is a structure holding a guard, and if the guard is canExit, it also holds information about the counting loop and its bounds.

7.2 Determinization of the Counting Automaton

The second step of the algorithm is the determinization of the counting automaton created from the input regex. The input of this part is an instance of the `Regexp::Derivatives` class, which holds all the information about the CA. The output of this part of the algorithm is an instance of a `CSA` class. The instance holds the resulting counting-set automaton. If the determinization is done on the fly, the instance does not hold the whole CsA from the start. However, it does carry all the information needed for the determinization, and the CsA is building as needed during the matching. The on-the-fly determinization can also gradually create whole CsA for some regexes and input text.

Computing the State Scope

The first step of the determinization part of the algorithm is to compute the scope of the states. The state scope is needed to compute the set of minterms of the set of counter guards defined in Equation 6.15. The state scope is also needed to compute the counting-set operator f of the newly created transitions defined in Equation 6.17.

The scope will be computed only for the reachable states. Therefore, the computation starts with the initial state. Then all the states reachable using the transitions from the currently explored state are added to a `statesToExplore vector`. All states from the vector are gradually explored. The computation ends when there is no state to explore.

The formal definition of the scope is in Definition 6.3.3. The scope of the states is computed using the `computeStateScopes` method. The method will explore all the outgoing transitions of the currently explored state. For each of the transitions, it also traverses all its operations. If there is the incr or the exit1 operation, it will add the corresponding counting loop of the operation to the scope of the target state of the transition. If there is an exit operation, the counting loop must be erased from the scope of the target state of the transition if it is there.

According to the definition of the scope, the **ID** operation should spread the current scope of its counting loop. However, during the translation of the regex into the CA, there

is no information about the counting loop that corresponds to the inserted **ID** operation in Equation 6.9 and Equation 6.11. Also, there has to be an implicit **ID** operation for all counting loops that are not used on the transition in some other operation. Therefore, the `computeStateScopes` method keeps track of all counter used on individual transitions. Then it gets all unused counting loops of the transition as a set difference of all counting loops of the regex (these are obtained in the translation of the regex to CA step of the algorithm) and all counting loops used on the transition. The method will create an **ID** operation for all the unused counting loops and add it to the transition. All the unused counting loops that are in the scope of the current state must spread the scope to the target state too. Such counting loops are computed as a set intersection of all counting loops that are in the scope of the current state and all unused counting loops (i.e., those with the implicit **ID** operation). The resulting scope of the target state is then a union of the already computed scope of the target state and all counting loops that are in the result of the intersection.

Computing the CsA States and Transitions

The computation of the CsA states and transitions is done by one primary method, named `getNextStateAndTransitions`. It implements all the steps described in the Generalized Subset Construction. It also uses some other methods representing the steps of the formal algorithm, namely `getCounterGuards`, `computeMinterms`, `checkSatisfiability`, and `getCsaTransitionOperator`.

`getCounterGuards`

This method will get the set of counter guards on the transition. The set is needed to compute the set of minterms $\Gamma_{R,\alpha}$ defined in Equation 6.15. According to the definition, it gets the counter guards for the given CsA state and the given bytemap class. It traverses all the outgoing transitions of the given state. For each transition, it will check if the given bytemap class is the same as the bytemap class of the transition. The transition can also have the bytemap class 256. Such transition can also be processed since the bytemap class 256 means any character. Other transitions are skipped.

Then, each of the transitions that fulfill the bytemap class condition is processed. That means to traverse all its counter guards. All the guards of the transition that are in the scope of the source state are added to the final set of the counter guards. The method also gets a set of all the counting loops of the used guard. The set of counting loops will be used later for optimizations.

`computeMinterms`

This method does the second step of the set of minterms $\Gamma_{R,\alpha}$ computation. It computes the minterms of the given set of counter guards. The algorithm is inspired by the algorithm for minterms computation introduced in [18]. The individual minterms are sets of counter guards.

The algorithm creates two sets. The first set contains the first counter guard, and the second contains the same counter guard but negated. These two sets that will be potentially extended represent the minterms. The sets are also added to the set of all current minterms. The method then traverses all the remaining counter guards. For each of the counter guards, it will try to extend all of the current minterms by the non-

negated and negated guard separately. I.e., there could be two minterms after this step, the original minterm extended by the non-negated guard and the original minterm extended by the negated guard. The non-negated guard is always added. The conjunction of all the guards in the minterm must remain satisfiable after the negated guard is added. There are only two situations when the conjunction could be unsatisfiable. The first situation is when the guard is `True`, its negated version is ¬`True`, which is always unsatisfiable. The second is the conjunction $\neg\text{CANINCR}_c \wedge \neg\text{CANEXIT}_c$, which can never be satisfied for non-empty sets of positive integers. In all other cases, the negated guard is also added to the minterm. The computation ends when all the guards are traversed.

`checkSatisfiability`

The method checks if all the operations of the given transition are consistent with the given minterm. I.e., there is no combination such as INCR_c operation and $\neg\text{CANINCR}_c$ guard. This check has to be done in order to get the $\Delta_{R,\alpha,\beta}$ set of CA transitions (defined in Equation 6.16) that are used to build the CsA transition.

The method gets two sets of guards as arguments. The first is the counter guards of the currently checked transition (i.e., φ_f in the definition). That is why the counter guards are explicitly saved to CA transitions as described in the Structure of the Resulting Counting Automaton. Thanks to that, the operations of the transition do not have to be traversed in this method to get the guards.

The second set of guards is the minterm, i.e., β in the definition. However, it is not the original minterm but a modified one. All of the guards in the minterm are negated. For example, $\neg\text{CANINCR}_c$ will become CANINCR_c, and CANINCR_c will become $\neg\text{CANINCR}_c$.

With the modification of the minterm, the satisfiability of the conjunction of these two sets of guards can then be checked using the set intersection. The conjunction could become unsatisfiable if there is the same guard in both sets, but one of them is negated (i.e., CANINCR_c in one set and $\neg\text{CANINCR}_c$ in the other set). Because all the guards of the minterm are negated before this method is called, it can be checked if there are two same guards. In such a case, the conjunction is unsatisfiable. Therefore, if the result of the set intersection is non-empty, the conjunction is unsatisfiable; otherwise, it is satisfiable. An example of how the method works is in Example 7.2.1.

Example 7.2.1. *Let $\beta = \{\neg\text{CANINCR}_c, \text{CANEXIT}_y\}$ be the minterm. According to the description above, its modified version will then be $\beta' = \{\text{CANINCR}_c, \neg\text{CANEXIT}_y\}$. Assume that the second argument of the* ***`checkSatisfiability`*** *method is the set of guards of the transition $\varphi_f = \{\text{CANINCR}_c, \text{CANEXIT}_y\}$. The method compute intersection $\beta' \cap \varphi_f = \{\text{CANINCR}_c\}$. The result of the intersection is non-empty, and, therefore, the conjunction should be unsatisfiable. This is right since the conjunction of $\neg\text{CANINCR}_c$ from the original minterm and CANINCR_c from the set of guards of the transition is unsatisfiable.*

`getCsaTransitionOperator`

This method implements Equation 6.17. It traverses all the operations of the given transition and creates the set of operations for CsA transition.

`getNextStateAndTransitions`

This is the main method of determinization algorithm. It gets all outgoing transitions for the given CsA state. It first computes the set of minterms $\Gamma_{R,\alpha}$ for the given CsA state R and

given the bytemap class α using the `getCounterGuards` and `computeMinterms` methods. The method then traverses the set of minterms. For each of the minterms, it creates its modified version according to the description above.

For each of the modified minterms, it traverses all outgoing transitions of the given CsA state. The CsA state is the set of CA states, so it traverses all outgoing transitions of all CA states from the CsA state. For each of the transitions, it checks if it can be taken. That means to check the bytemap class of the transition, which must be equal to the given bytemap class α or it must be 256 (i.e., it matches any character). Also, the conjunction of the set of guards of the transition and the current minterm must be satisfiable. That is checked using the `checkSatisfiability` method called with the modified version of the minterm and the set of guards of the transition. If the transition can be taken, its target state will become part of the target CsA state. If the target CA state is final, the method also adds the corresponding final state condition to the set of the CsA state final conditions. Also, it computes the CsA operation for each of the transitions that can be taken.

The last step is to save the information about the newly computed transitions and states. The following information will be saved for the transitions:

- the source state of the transition which is the given state R,
- bytemap class which is the given bytemap class α,
- the set of used counting loops computed by the `getCounterGuards` method,
- the set of operations computed by the `getCsaTransitionOperator` method,
- the target state, which is obtained as the set of target CA states of the used transitions.

If any of the CA states that form the target CsA state is the final state, the target CsA state will also be saved as final.

The Structure of the Counting-Set Automaton

The counting-set automaton is the vector, which will be called the *automaton vector* for clarity. Its index corresponds to the numerical representation of the source state. The value of the automaton vector stores information about the possible transitions. The information is stored as a vector, which will be called the *state vector*. The index of the state vector corresponds to the bytemap for which the transitions can be taken. The values of the state vector are pairs of used counting loops and the vector of transitions itself. The index to the vector of transitions is based on the used counter guards (the computation of the index will be described further in the text). The value on the index is then a pair of the set of counter operations and the target state.

The structure of the automaton is optimized so that the matching loop can work efficiently with it. The first optimization is to use the numerical representation of the states. Then the whole automaton can be saved in the vector (the automaton vector), which is accessed by the index (i.e., the numerical representation of the state). The vector provides a constant time access by the index. Constant time access is crucial. Containers like the `unordered_map` provide average constant complexity. However, the worst-case complexity is linear, and the run-time difference between these two is significant.

The state vector holds pairs of the used counting loops and the transition vectors. The counting loops are used to compute the index to the transitions vector, so it is

not necessary to work with all counting loops of the regex. The index is computed by the `computeGuardIndexes` method described below. The computation of the index is used because the matching loop then does not have to traverse all the possible transitions (and check its guards) to find the only one that will be used. The transition itself holds information about the counter operations that will be used when it is taken and the target state of the transition.

Also, the state vector and the transition hold their values as `std::pair` rather than `std::tuple` because the `pair` proves to be faster. Note that the `pair` can not be used to hold more than two values, whereas the `tuple` can hold multiple values. However, in the described structure, it is enough to have only pairs of values, and, therefore, it can take advantage of the `std::pair` speed.

`computeGuardIndexes`

This method is used to compute the index of the transition in the transitions vector based on its guards. This optimization is based on the `C#` implementation of [48]. Each counter-memory can be described by one of the following states:

- *LOW* denotes the counter-memory where all its values are below the lower bound of the counting loop,
- *HIGH* denotes the counter-memory with a single value which is equal to the upper bound of the counting loop,
- *MIDDLE* denotes the rest.

Based on that, each counter guard can be satisfied by the counter-memory in some states. More specifically, the CanIncr guard is satisfied by counter-memory in state LOW and MIDDLE since there is at least one value lower than the upper bound of the counting loop in both of the states that can be incremented. The CanExit guard is satisfied by the counter-memory in the state MIDDLE and HIGH since there is at least one value greater than the lower bound of the counting loop. For the conjunction of the guards, the states that satisfy individual guards can be viewed as sets. The states satisfying the conjunction are obtained as the intersection of these sets. For example, the conjunction CanIncr $\wedge$ CanExit can be satisfied by the counter-memory in state $\{\text{LOW, MIDDLE}\} \cap \{\text{MIDDLE, HIGH}\} = \{\text{MIDDLE}\}$.

Each of the LOW, MIDDLE, and HIGH states is internally represented as a number, specifically one, two, and three, respectively. The index of the transition in the transition vector is then computed using bit operations. The method prepares an empty vector for indexes at the start and traverses the vector of used counting loops. The counting loop is represented by the number of the current iteration. It then gets the numerical representation of the counter state based on the guards of the transition and updates the vector of indexes. If two states satisfy the guard (for example, LOW and MIDDLE), they must be used separately for the index computation, which results in two indexes being computed.

The update of the index is performed in two ways based on the emptiness of the vector of indexes. If the vector is empty, it adds a new index based on the current counter number. The index is computed as $\texttt{state\_number} \ll \texttt{counting\_loop\_number} * 2$, where $\ll$ represents left bit shift operation. Since each of the numerical representations of LOW, MIDDLE, and HIGH can be represented by two bits, the $\texttt{counting\_loop\_number} * 2$ part

of the index computation ensures that each counting loop has its own two bits in the binary representation of the index. If the vector of indexes already contains some computed indexes, the method traverses them all and updates them by the newly computed index using logical `OR` operation. The logical `OR` operation ensures that all the values representing the already processed counters remain unchanged. An example of the index computation is in Example 7.2.2.

Example 7.2.2. *Assume that there are two counting loops represented by numbers zero and one. Also, assume that the guards of the counting loops are* $\textsc{CanIncr}_0 \wedge \textsc{CanExit}_0$ *and* $\textsc{CanIncr}_1 \wedge \neg\textsc{CanExit}_1$*. The computation starts with the counting loop zero. The state that satisfies its guard is the MIDDLE state (which is represented by number two). Therefore, the index is computed as* $2 \ll 0 * 2 = 2$*, which is* 10 *in binary. The index is added to the vector, and the computation continues with the counting loop one. The state that satisfies its guard is the LOW state (which is represented by number one). The index for this counting loop is then computed as* $1 \ll 1 * 2 = 4$*, which is* 0100 *in binary. However, since there is already one index computed, these two must be combined using the logical* `OR` *operation. The resulting index is then* $10|0100 = 0110$*. The counting loop zero is represented on the lower two bits of the result, and the counting loop one is represented on the higher two bits of the result.*

7.3 Matching With the Counting-Set Automaton

The matching is done using the CsA created using previous steps of the algorithm. There are two kinds of matching implemented. The first is the so-called full match, which matches only the whole input string, and the second is the so-called partial match, which matches the string anywhere on the line. However, the matching algorithm itself is the same for both of these kinds.

The difference between the two kinds of matching is in the way the input regex is processed. For the full match, the regex is processed as it is entered on the input. Then the CA is created and determinized. For the partial match, the regex must be updated. The new regex must find the original regex anywhere on the line. This is accomplished by adding the `.*` as a prefix and a suffix to the original regex. The newly created regex can read any characters before and after the original regex, and the original regex can be found anywhere on the line. The algorithm treats the updated regex the same as in the full match. It is just the regex itself making the difference between the full match and the partial match.

The main matching algorithm iterates over the input text character by character. For each of the characters, it gets the bytemap class using the pre-computed bytemap class array. The character is used as the index to the bytemap class array; the value is then the bytemap class for the currently processed character.

The matching algorithm then works with the CsA. However, as the CsA is built on the fly, the matching algorithm needs to know if the desired part of the CsA was already computed or not. For that, it uses a vector, where the index is the numerical representation of the source state. The value is another vector, where the index is the bytemap class number. The values of the inner vector are booleans that determine if the state and bytemap class combination was already processed. Therefore, the matching algorithm checks if the value on the index `[state_number][bytemap_class]` is true or false. If it is false, the matching algorithm calls the `getNextStateAndTransitions` method with the current

state and current bytemap class. If the value is true, the outgoing transition for the current state and bytemap class combination was already computed and will be used without any additional computation. After that step, the matching loop also checks if there are any outgoing transitions computed. There could be zero outgoing transitions meaning that the match fails.

The on-the-fly determinization has an advantage on the input texts, for which only a part of the CsA is used. In such a case, the unused parts are not computed, as opposed to the CsA pre-computing, where the whole CsA is computed before the matching starts. However, in the worst case, even the on-the-fly determinization can end up computing the whole CsA.

The matching loop then gets the above-described state vector for the state and bytemap class combination as the result of processing the combination or accessing the cache. As the next step, it is necessary to compute the index to the vector of transitions based on the used counting loops and current counter-memory values. The index is obtained similarly to the index computation during the determinization described above. It also uses the bit shifts and logical `OR` operation. The difference is how the LOW, MIDDLE or HIGH state of the counter is determined. In the determinization step, it is obtained from the guards of the transition, which means that it gets the states of the counter-memory that satisfy the guard. In the matching loop, the counter-memory states are obtained based on the current values of the counter-memory, which means that it gets the information about what guards the current counter-memory satisfies. Therefore, the matching loop checks the current state of the counter-memory for each of the used counting loops and gets the index to the transitions vector.

After the index to the transitions vector is obtained, the matching loop checks if there is any transition on that index. That means checking if the index is not out of the bounds of the vector. If it is not, it also checks if the transition is not just a default one. The default transitions are on those indexes of the transitions vector, for which there is no transition to be taken, which means that the match fails. This corresponds to the situation when the automaton can not move the next state with the current values of the counter-memory (i.e., there was not enough repetition of the pattern).

When the valid transition is found, the last step of the matching loop is to update the counter-memory by the corresponding operations of the transition. The individual operations of the transition are applied to the counter-memory separately, and the resulting updated counter-memory is the union of such individual updates. However, all the combinations of operations, except the Noop and Incr combination, can be implemented in constant time since the Rst and Rst1 create a single element set. The Noop and Incr combination can be implemented in time linear to the size of the resulting sets of these operations.

The individual operations with the counter-memory are implemented according to the description in the Counting-Set Automata. The used structure is the vector, which holds pairs of the offset, and `std::deque`, which is used as the queue. The index in the vector is the numerical representation of the counting loop.

When the counter-memory is updated, the matching loop continues with the next character from the input text. It repeats all those described steps until the whole input text is traversed. After that, it is checked if the lastly visited state is final or not. If it is not final, the match failed. If it is final, it is also necessary to check if at least one of its final conditions holds. The final state conditions are held in the `std::set` container sorted by the guard, so the $\top$ guard, which means that the state is unconditionally final, is the first.

The algorithm traverses all the final state conditions, checking them one by one. When one of the conditions is satisfied, the match is successful. Note that since the set of guards is sorted, if there is some $\top$ guard, it will be checked first, and no other guards will be checked. If there are only the CanExit guards, the algorithm also checks the current state of the counter-memory to determine if the guard is satisfied or not. If none of the final state conditions is satisfied, the match is unsuccessful.

7.4 Tests

The set of tests is implemented in `Python`. The tests are written to cover the basic functionality of the individual steps of the translation regex into the CA, determinization of the CA, and also the matching itself. The test runs the corresponding part of the algorithm on a set of regexes and checks if the output matches the expected one. The implemented matching algorithm does not print the output of the individual steps by default. The `#if defined` directive and compilation with the corresponding flag is used to enable or disable the part of code that prints the output of the desired part of the algorithm. The parts covered by tests are the following:

1. *equations test* that covers the output of the `getEquationTypeAndOperands` method,
2. *normalization test* that covers the normalization of the input regex,
3. *derivatives test* that covers the whole first step of the algorithm, i.e., translation of the input regex into the CA,
4. *CA final states test* that covers the identification of the final states and their conditions in the CA,
5. *scope test* that covers the computation of the scope of the counters,
6. *gamma set test* that covers the computation of the $\Gamma_{R,\alpha}$ set,
7. *determinization test* that covers the determinization of the CA (i.e., it checks if the created CsA is correct),
8. *CsA final states test* that covers the identification of the final states and their conditions in the CsA,
9. *matching test* that covers the full match type of matching,
10. *unanchored matching test* that covers the partial match type of matching.

Each of the tests, except the matching tests, contains the set of patterns and their corresponding correct outputs. The main script of the test firstly compiles the RE2 with the corresponding flag to get the output of the tested part printed. Then it prepares the source code with the patterns, runs it, and compares the output with the predefined correct output. If they match, it reports the test of the current pattern as successful. If they do not match, it reports it as an unsuccessful test and prints out the expected and actual output so it can be further checked.

The matching tests have only the patterns defined, not the outputs. The testing script then prepares the source code with the patterns so that the patterns are matched by

the newly implemented CsA-based algorithm and the original RE2 algorithm. The outputs are then compared, and the test is successful if the CsA-based algorithm has the same result of matching as the original algorithm.

The main test script cleans out the created source codes after the tests are finished. It also runs `make clean` for the RE2 so that the next compilation will be done correctly with the current flag. How to use the test is described in Appendix B.

7.5 Benchmarks

Similar to the tests script, there is also a Python script that runs the benchmarks for the compilation of the input regex into the CA and the determinization of the CA. These benchmarks are run on 37 regexes that were challenging for the `C#` implementation. The result of these benchmarks is discussed in the Translating of the Regex Into the CA and Determinization of the CA.

The benchmarks can be run for regexes used both for the full match and partial match type of matching. The main script runs the compilation of the input regex into the CA or the determinization on each of the defined regexes. The output of the algorithm compiled with the corresponding flag for benchmarks is the time it takes to create the CA or the CsA. The Python script collects the times, prints them for each of the patterns. It also prints the sum of the times at the end of the run. Also, it has a defined timeout, after which it kills the computation and reports it as the timeout. How to use the benchmarks script is in Appendix C.

The benchmarks of the matching itself are run within the benchmark tool of [48].

Chapter 8

Experimental Evaluation

This Chapter discusses the experimental evaluation of the algorithm. It is divided into two parts. The first discusses only the translation of the regex into the CA and the determinization of the CA parts of the algorithm. These two parts of the algorithm are compared only with the `C#` implementation named `CA` from [48]. The reason is that it is the only implementation using the CsA-based matching and, therefore, the only implementation that creates and determinizes the counting automaton. The second part discusses the experimental evaluation of the matching itself.

8.1 Translating of the Regex Into the CA and Determinization of the CA

These parts of the implementation were tested only on more complex regexes from the benchmark of [48] that were hard to determinize for the `C#` implementation.

	`CA`		RE2 (CsA)		
	NCA	CsA	NCA	CsA	CsA without timeouts
mean	10 966	73 433	8	62 709	11 136
median	254	11 338	4	1 990	1 050
timeouts	0	14	0	5	N/A

Table 8.1: Experimental evaluation of RE2 (with CsAs) implementation and `CA` (times are given in milliseconds). NCA denotes the time from the loading of input regex to the counting automaton being created. CsA denotes the time of the determinization of the counting automaton. CsA without timeouts denotes the time of the determinization, which was run only on regexes on which any of the two implementations do not suffer from timeout. The timeout was set to 1 800 seconds.

The result in Table 8.1 shows that the RE2 implementation is faster for both translations of the regex into counting automata and determinization of the counting automata. The determinization part of the RE2 implementation takes a long time on some regexes on which the CA implementation suffers from timeout. There is also the result of the RE2 implementation on regexes on which the CA does not suffer from timeout to stretch out the difference.

8.2 Regular Expression Pattern Matching

The matching benchmarks were run on a set of 2320 regexes from the benchmark of [48]. The actual comparison of matching times was made based on 2148 benchmarks, where the CsA-based RE2 algorithm found the correct number of matches. The results of the original RE2 implementation were considered as a reference for the number of matches. Also, each of the individual comparisons was made based only on those regexes, for which both of the compared tools get the correct number of matches. Note that the comparison with grep and also the overall statistics are done on a smaller set of regexes. It is because grep uses different semantics of regexes, causing it to quit after reading only several characters and reporting zero matches on such regexes. Therefore, regexes for which the time and the number of matches of grep were zero, and the other tools find a match or their matching time was larger than zero (i.e., the grep quit after reading just several characters while the other tools try to find a match in the whole input text), were not considered.

The tools were run in the settings, where their output is the number of lines on which the match was found. If there are no explicit anchors, the matched string could be anywhere on the line. If there is a start line (`^`) or end line (`$`) anchor, the match has to start at the beginning or at the end of the line, respectively. If both anchors are used in a single regex, the whole line has to be matched.

The main focus of the matching benchmarks was to compare CsA-based matching implemented in `C#` and in `C++`. Such comparison determines if the `C++` implementation speeds up the original CsA-based matching implementation or not. The state-of-the-art matchers grep and the original RE2 is also included in the comparison.

The result of the comparison between the CsA-based RE2 implementation and the other tools in the form of scatter plots is in Figure 8.1. The plot in Figure 8.1(c) shows the result of a comparison between the `C#` and `C++` implementation of CsA-based matching. The `C++` implementation did indeed speed up the CsA-based algorithm, as it is faster in 1908 out of 2145 benchmarks. For some of the benchmarks, the matching time of both of the tools was significantly higher. However, it stays under one second for most of the benchmarks.

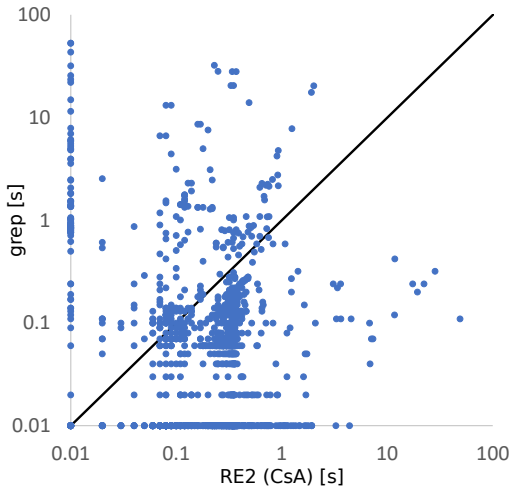
The comparison between the CsA-based RE2 implementation and the other tools, grep, and original RE2 is in Figure 8.1(a) and Figure 8.1(b), respectively. Even though the grep and original RE2 win more often, there are clearly regexes, for which the CsA-based RE2 implementation is faster than these two. Those are the regexes with the counting loops, often with higher bounds. Considering the plot in Figure 8.1(c), it is clear that the CsA-based matching has more stable matching times than the original RE2 and grep.

	RE2	grep	CA	RE2 (CsA)
mean	0.153	0.537	0.480	0.399
median	0	0.06	0.43	0.28
std. deviation	1.614	3.175	0.207	1.739
timeouts	1	10	2	2

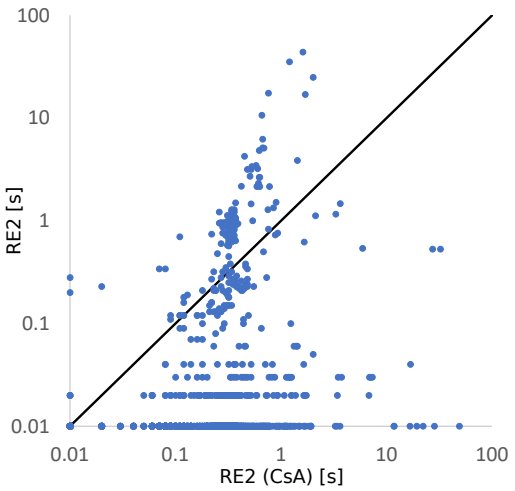
Table 8.2: The benchmarks statistic of the individual tools. Times are given in seconds, and the timeout was set to 60 seconds. The supplied times are based on 1625 benchmarks on which none of the tools suffers from error or timeout, and the regex suits the grep semantics. For the timeout statistics, the regexes for which the tools suffer from timeout were also added, resulting in a set of 1640 regexes being used for the timeout statistics.

Table 8.2 contains the statistic of matching times of individual tools. Comparing the two CsA-based algorithms, the one implemented within RE2 is faster. Comparing the CsA-based RE2 with grep, the mean is better for the CsA-based RE2. Even though the grep wins most of the time (as shown in Figure 8.1(a)), there are regexes on which the CsA-based RE2 is significantly faster, making the overall mean better for the CsA-based RE2. Grep also suffers from timeout the most of all compared tools.

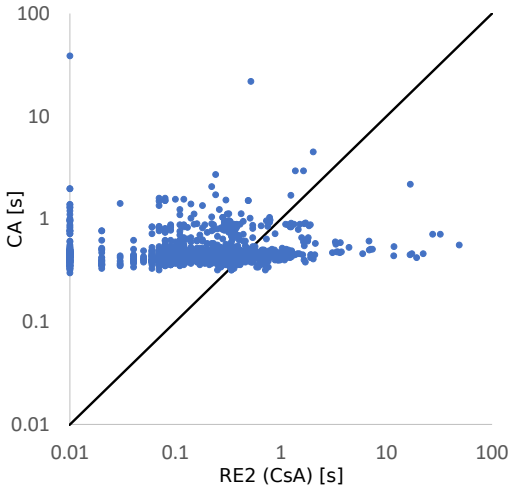
The original RE2 is the fastest among all the tools. It also suffers from timeout the least. Considering the plot in Figure 8.1(b), the speed of original RE2 still drops on regexes with counting loops with higher bounds on which the CsA-based RE2 is faster than the original one. The usage of the CsA-based algorithm in RE2 is determined by a constructor parameter. This allows the developer to choose the CsA-based algorithms or the original algorithms. If the developer has control over the regexes (i.e., the regexes are not supplied by the user), the developer can use the CsA-based algorithm for the counting-heavy regexes and the original algorithms for the rest of the regexes, where the numerous RE2 optimizations will do faster.



(a) The comparison of matching times between CsA-based matching implemented within RE2 and grep. RE2 with CsA-based matching is faster in 269 benchmarks from the total number of 1629 benchmarks (the set of benchmarks is smaller for the grep because of its different semantic of regexes). The speed is the same in 30 benchmarks.



(b) The comparison of matching times between CsA-based matching implemented within RE2 and the original RE2. RE2 with CsA-based matching is faster in 139 benchmarks from the total number of 2148 benchmarks. The speed is the same in 109 benchmarks.



(c) The comparison of matching times between CsA-based matching implemented within RE2 and CA, which also implements CsA-based matching. RE2 with CsA-based matching is faster in 1908 benchmarks from the total number of 2145 benchmarks. The speed is the same in 11 benchmarks.

Figure 8.1: Graphs with the results of the comparison between CsA-based matching implemented in RE2 with grep, original RE2, and CA, which also implements the CsA-based matching.

Chapter 9

Conclusion

The goal of this thesis was to implement efficient counting-set-automata-based regular expression (regex) matching. The counting-set automata approach for regex matching is introduced in [48], where it is also shown that such approach is efficient for regexes with bounded repetition operator and can outperform state-of-the-art matchers on such regexes.

The implementation was done within the state-of-the-art matcher RE2, which allowed the usage of its already implemented parts. The determinization and matching part of the algorithm contains two optimizations. The first allows to effectively choose the right transition based on the current state of the counter-memory. The second is the usage of the on-the-fly determinization.

The implementation was experimentally evaluated and compared with the `C#` implementation of [48], grep, and the original RE2. The speed of translation of the input regex into the counting automaton and determinization of the automaton was compared for the CsA-based RE2 and `C#` implmenentation. The RE2 implementation shows a speed-up in both translations of the input regex and determinization of the counting automata.

In terms of the speed of the matching, the new CsA-based RE2 implementation outperforms the original `C#` implementation on most of the regexes. Compared with grep, the CsA-based RE2 was slower on more regexes. However, the mean of the matching time was better for the CsA-based RE2. Also, grep suffers from timeout more than the CsA-based RE2. The original RE2 was faster for most of the regexes, and it also has the lowest mean time upon all the tools.

The implementation contains the above-described optimization, which helped to outperform the original implementation on most of the regexes, and the grep and the original RE2 on regexes with bounded repetition with higher bounds. However, the numerous advanced optimizations of the grep and the original RE2 make them faster on the rest of the regexes. The first option is to re-implement such optimizations and used them in the CsA-based RE2 algorithm. Since the new algorithm is implemented within the original RE2, there is a second option. The usage of the CsA-based algorithm in RE2 is currently determined by the constructor parameter. Currently, the developer can choose if it is suitable for a specific regex to use the CsA-based algorithm or the original algorithm. That way, the advantage of the counting-set automata can be used for the regexes with bounded repetition, and the numerous optimizations of the original algorithm can make it faster on the other regexes. It would be even better to automatize this concept and choose the appropriate algorithm based on the input regex.

Bibliography

[1] *GNU Grep 3.5* [online]. [cit. 2021-05-16]. Available at: https://www.gnu.org/software/grep/manual/grep.html.

[2] Local languages and the Berry-Sethi algorithm. *Theoretical Computer Science*. 1996, vol. 155, no. 2, p. 439 – 446. DOI: https://doi.org/10.1016/0304-3975(95)00104-2. ISSN 0304-3975. Available at: http://www.sciencedirect.com/science/article/pii/0304397595001042.

[3] *Outage Postmortem* [online], 20. july 2016 [cit. 2020-12-27]. Available at: https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016.

[4] Abdulla, P. A., Krcal, P. and Yi, W. R-Automata. In: Breugel, F. van and Chechik, M., ed. *CONCUR 2008 - Concurrency Theory*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 67–81. ISBN 978-3-540-85361-9.

[5] Aho, A. V. and Corasick, M. J. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. june 1975, vol. 18, no. 6, p. 333–340. DOI: 10.1145/360825.360855. ISSN 0001-0782. Available at: https://doi.org/10.1145/360825.360855.

[6] Antimirov, V. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*. 1996, vol. 155, no. 2, p. 291 – 319. DOI: https://doi.org/10.1016/0304-3975(95)00182-4. ISSN 0304-3975. Available at: http://www.sciencedirect.com/science/article/pii/0304397595001824.

[7] Baldwin, A. *Regular Expression Denial of Service affecting Express.js* [online], 16. june 2016 [cit. 2020-12-27]. Available at: http://web.archive.org/web/20170116160113/https://medium.com/node-security/regular-expression-denial-of-service-affecting-express-js-9c397c164c43.

[8] Bardin, S., Finkel, A., Leroux, J. and Petrucci, L. FAST: Acceleration from theory to practice. *International Journal on Software Tools for Technology Transfer*. september 2008, vol. 10, p. 401–424. DOI: 10.1007/s10009-008-0064-3.

[9] Berry, G. and Sethi, R. From regular expressions to deterministic automata. *Theoretical Computer Science*. 1986, vol. 48, p. 117 – 126. DOI: https://doi.org/10.1016/0304-3975(86)90088-5. ISSN 0304-3975. Available at: http://www.sciencedirect.com/science/article/pii/0304397586900885.

[10] Björklund, H., Martens, W. and Timm, T. Efficient Incremental Evaluation of Succinct Regular Expressions. In: *Proceedings of the 24th ACM International on*

Conference on Information and Knowledge Management. New York, NY, USA: Association for Computing Machinery, 2015, p. 1541–1550. CIKM '15. DOI: 10.1145/2806416.2806434. ISBN 9781450337946. Available at: https://doi.org/10.1145/2806416.2806434.

[11] Brzozowski, J. A. Derivatives of Regular Expressions. *J. ACM*. New York, NY, USA: Association for Computing Machinery. october 1964, vol. 11, no. 4, p. 481–494. DOI: 10.1145/321239.321249. ISSN 0004-5411. Available at: https://doi.org/10.1145/321239.321249.

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

[13] Caron, P., Champarnaud, J.-M. and Mignot, L. Partial Derivatives of an Extended Regular Expression. In: Dediu, A.-H., Inenaga, S. and Martín Vide, C., ed. *Language and Automata Theory and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, p. 179–191. ISBN 978-3-642-21254-3.

[14] Champarnaud, J.-M. and Ziadi, D. Computing the Equation Automaton of a Regular Expression in O(s2) Space and Time. In: Amir, A., ed. *Combinatorial Pattern Matching*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, p. 157–168. ISBN 978-3-540-48194-2.

[15] Chapman, C. and Stolee, K. T. Exploring Regular Expression Usage and Context in Python. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2016, p. 282–293. ISSTA 2016. DOI: 10.1145/2931037.2931073. ISBN 9781450343909. Available at: https://doi.org/10.1145/2931037.2931073.

[16] Charras, C. and Lecroq, T. *Handbook of Exact String Matching Algorithms*. King's College Publications, 2004. ISBN 0954300645.

[17] Cheng, K. T. and Krishnakumar, A. S. Automatic Functional Test Generation Using the Extended Finite State Machine Model. In: *Proceedings of the 30th International Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, 1993, p. 86–91. DAC '93. DOI: 10.1145/157485.164585. ISBN 0897915771. Available at: https://doi.org/10.1145/157485.164585.

[18] D'Antoni, L. and Veanes, M. Minimization of Symbolic Automata. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2014, p. 541–553. POPL '14. DOI: 10.1145/2535838.2535849. ISBN 9781450325448. Available at: https://doi.org/10.1145/2535838.2535849.

[19] 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*. New York, NY, USA: Association for Computing Machinery, 2019, p. 1256–1258. ESEC/FSE 2019. DOI: 10.1145/3338906.3342509. ISBN 9781450355728. Available at: https://doi.org/10.1145/3338906.3342509.

[20] Davis, J. C., Coghlan, C. A., Servant, F. and Lee, D. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018, p. 246–256. ESEC/FSE 2018. DOI: 10.1145/3236024.3236027. ISBN 9781450355735. Available at: https://doi.org/10.1145/3236024.3236027.

[21] Davis, J. C., Michael IV, L. G., Coghlan, C. A., Servant, F. and Lee, D. 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*. New York, NY, USA: Association for Computing Machinery, 2019, p. 443–454. ESEC/FSE 2019. DOI: 10.1145/3338906.3338909. ISBN 9781450355728. Available at: https://doi.org/10.1145/3338906.3338909.

[22] Fischer, S., Huch, F. and Wilke, T. A Play on Regular Expressions: Functional Pearl. In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: Association for Computing Machinery, 2010, p. 357–368. ICFP '10. DOI: 10.1145/1863543.1863594. ISBN 9781605587943. Available at: https://doi.org/10.1145/1863543.1863594.

[23] Friedl, J. *Mastering Regular Expressions*. 3rd ed. O'Reilly Media, Inc., 2006. ISBN 0596528124.

[24] Gelade, W., Gyssens, M. and Martens, W. Regular Expressions with Counting: Weak versus Strong Determinism. *SIAM Journal on Computing*. 2012, vol. 41, no. 1, p. 160–190. DOI: 10.1137/100814196. Available at: https://doi.org/10.1137/100814196.

[25] Glushkov, V. M. THE ABSTRACT THEORY OF AUTOMATA. *Russian Mathematical Surveys*. IOP Publishing. oct 1961, vol. 16, no. 5, p. 1–53. DOI: 10.1070/rm1961v016n05abeh004112. Available at: https://doi.org/10.1070/rm1961v016n05abeh004112.

[26] Goyvaerts, J. *Runaway Regular Expressions: Catastrophic Backtracking* [online], 22. december 2019 [cit. 2021-01-11]. Available at: https://www.regular-expressions.info/catastrophic.html.

[27] Graham Cumming, J. *Details of the Cloudflare outage on July 2, 2019* [online], 12. july 2019 [cit. 2020-12-27]. Available at: https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

[28] Haertel, M. *Why GNU grep is fast* [online], 21. august 2010 [cit. 2021-05-18]. Available at: https://lists.freebsd.org/pipermail/freebsd-current/2010-August/019310.html.

[29] Holík, L., Lengál, O., Saarikivi, O., Turoňová, L., Veanes, M. et al. Succinct Determinisation of Counting Automata via Sphere Construction. In: Lin, A. W., ed. *Programming Languages and Systems*. Cham: Springer International Publishing, 2019, p. 468–489. ISBN 978-3-030-34175-6.

[30] Hovland, D. Regular Expressions with Numerical Constraints and Automata with Counters. In: Leucker, M. and Morgan, C., ed. *Theoretical Aspects of Computing - ICTAC 2009*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, p. 231–245. ISBN 978-3-642-03466-4.

[31] Ilie, L. and Yu, S. Follow automata. *Information and Computation*. 2003, vol. 186, no. 1, p. 140 – 162. DOI: https://doi.org/10.1016/S0890-5401(03)00090-7. ISSN 0890-5401. Available at: http://www.sciencedirect.com/science/article/pii/S0890540103000907.

[32] Kilpeläinen, P. and Tuhkanen, R. Regular Expressions with Numerical Occurrence Indicators - preliminary results. In: *Proceedings of the Eighth Symposium on Programming Languages and Software Tools*. Kuopio, Finland: University of Kuopio, Department of Computer Science, 2003, p. 163–173. SPLST'03.

[33] Lombardy, S. and Sakarovitch, J. Derivatives of rational expressions with multiplicity. *Theoretical Computer Science*. 2005, vol. 332, no. 1, p. 141 – 177. DOI: https://doi.org/10.1016/j.tcs.2004.10.016. ISSN 0304-3975. Available at: http://www.sciencedirect.com/science/article/pii/S0304397504007054.

[34] Meduna, A. *Automata and languages : theory and applications*. London: Springer, 2000. ISBN 1-85233-074-0.

[35] Navarro, G. and Raffinot, M. New techniques for regular expression searching. *Algorithmica*. Springer International Publishing. 2005, p. 89–116. Available at: https://doi.org/10.1007/s00453-004-1120-3.

[36] Neuburger, S. Pattern Matching Algorithms: An Overview. Department of Computer Science The Graduate Center, CUNY. september 2009.

[37] OWENS, S., REPPY, J. and TURON, A. Regular-expression derivatives re-examined. *Journal of Functional Programming*. Cambridge University Press. 2009, vol. 19, no. 2, p. 173–190. DOI: 10.1017/S0956796808007090.

[38] Russ, C. *Regular Expression Matching Can Be Simple And Fast: (but is slow in Java, Perl, PHP, Python, Ruby, ...)* [online]. January 2007 [cit. 2021-01-14]. Available at: https://swtch.com/~rsc/regexp/regexp1.html.

[39] Russ, C. *Regular Expression Matching in the Wild* [online]. March 2010 [cit. 2021-18-05]. Available at: https://swtch.com/~rsc/regexp/regexp3.html.

[40] Saarikivi, O., Veanes, M., Wan, T. and Xu, E. Symbolic Regex Matcher. In: Vojnar, T. and Zhang, L., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, p. 372–378. ISBN 978-3-030-17462-0.

[41] Shiple, T. R., Kukula, J. H. and Ranjan, R. K. A comparison of Presburger engines for EFSM reachability. In: Hu, A. J. and Vardi, M. Y., ed. *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, p. 280–292. ISBN 978-3-540-69339-0.

[42] Sipser, M. Introduction to the Theory of Computation. *SIGACT News*. New York, NY, USA: Association for Computing Machinery. march 1996, vol. 27, no. 1, p. 27–29. DOI: 10.1145/230514.571645. ISSN 0163-5700. Available at: https://doi.org/10.1145/230514.571645.

[43] Smith, R., Estan, C. and Jha, S. XFA: Faster Signature Matching with Extended Automata. In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. 2008, p. 187–201. DOI: 10.1109/SP.2008.14.

[44] Smith, R., Estan, C., Jha, S. and Siahaan, I. Fast Signature Matching Using Extended Finite Automaton (XFA). In: Sekar, R. and Pujari, A. K., ed. *Information Systems Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 158–172. ISBN 978-3-540-89862-7.

[45] Spencer, H. A Regular-Expression Matcher. In: *Software Solutions in C*. USA: Academic Press Professional, Inc., 1994, p. 35–71. ISBN 0126323607.

[46] Thompson, K. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. june 1968, vol. 11, no. 6, p. 419–422. DOI: 10.1145/363347.363387. ISSN 0001-0782. Available at: https://doi.org/10.1145/363347.363387.

[47] Turonova, L., Holik, L., Lengal, O., Saarikivi, O., Veanes, M. et al. *Regex Matching with Counting-Set Automata*. MSR-TR-2020-31. Microsoft, September 2020. Available at: https://www.microsoft.com/en-us/research/publication/counting-set-automata/.

[48] Turoňová, L., Holík, L., Lengál, O., Saarikivi, O., Veanes, M. et al. Regex Matching with Counting-Set Automata. *Proc. ACM Program. Lang.* New York, NY, USA: Association for Computing Machinery. november 2020, vol. 4, OOPSLA. DOI: 10.1145/3428286. Available at: https://doi.org/10.1145/3428286.

[49] Veanes, M., de Halleux, P. and Tillmann, N. Rex: Symbolic Regular Expression Explorer. In: *2010 Third International Conference on Software Testing, Verification and Validation*. April 2010, p. 498–507. DOI: 10.1109/ICST.2010.15. ISSN 2159-4848. Available at: https://doi.org/10.1109/ICST.2010.15.

[50] Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G. 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, February 2019, p. 631–648. ISBN 978-1-931971-49-2. Available at: https://www.usenix.org/conference/nsdi19/presentation/wang-xiang.

[51] Wankadia, P. *WhyRE2* [online], 18. june 2020 [cit. 2021-05-18]. Available at: https://github.com/google/re2/wiki/WhyRE2.

[52] Češka, M., Vojnar, T., Smrčka, A. and Rogalewicz, A. *Teoretická informatika: Studijní text*. Faculty of Information Technology, Brno University of Technology, 2020.

Appendix A

CD Content

The attached CD contains the following items:

- **re2-master** folder, which contains source files of RE2, including the new files for CsA-based matching (the `derivatives.h`, `derivatives.cc`, `csa.h`, and `csa.cc` files),
- **pythonScripts** folder, which contains all Python scripts for tests, and benchmarks of the automaton creation part of the algorithm,
- **install.sh** script that compiles the RE2 so it can be used in `C++` code,
- **README.md** file, which contains information about how to compile RE2 and how to use the CsA-based matching in `C++` code,
- **find_lines_csa** Linux executable file, it takes two arguments (regex and file) and prints out the number of lines containing matching string,
- **find_lines_csa.exe** Windows executable file, it takes two arguments (regex and file) and prints out the number of lines containing matching string,
- **DT_xhorky23.pdf** the text of this thesis,
- **text** folder, which contains LaTeX source files.

Appendix B

How to Run Tests

The test scripts are written in Python. The tests are supposed to be run on *Linux* as they compile the RE2 library with corresponding flags. They will not run on *Windows* directly but can be run using the *WSL*. The required version of Python is `3.6`. The main test script (`runTests.py`) is located in the `pythonScripts` folder (see Appendix A). The script has to be run from the `pythonScripts` folder as it accesses different paths and expects to be run from there. If the script is run without any argument, it will run all tests. It can also be used with an option `-t` or `--tests` followed by the test names. In such a case, it runs the specified tests. The test names are the following:

- `equation` to run the *equations tests*,
- `normalization` to run the *normalization tests*,
- `derivatives` to run the *derivatives tests*,
- `ca_final_states` to run the *CA final states tests*,
- `scope` to run the *scope tests*,
- `gamma_set` to run the *gamma set tests*,
- `determinization` to run the *determinization tests*,
- `csa_final_states` to run the *CsA final states tests*,
- `matching` to run the *matching tests*,
- `matching_unanchored` to run the *unanchored matching tests*.

See Section 7.4 for more information about the individual tests.

Each of the individual tests compiles the RE2 library with the corresponding flag to print the desired output. It then runs it on a set of regexes and compares the program output with the expected one. It prints out info about the pattern that is tested with the result of the test. If the test fails, it also prints the program output and the expected output.

Appendix C

How to Run Automaton Creation Benchmarks

The benchmark scripts are written in Python. The benchmarks are supposed to be run on *Linux* as they compile the RE2 library with corresponding flags. They will not run on *Windows* directly but can be run using the *WSL*. The required version of Python is `3.6`. The main test script (`runBenchmarks.py`) is located in the `pythonScripts` folder (see Appendix A). The script has to be run from the `pythonScripts` folder as it accesses different paths and expects to be run from there. If the script is run without any option, it will run all benchmarks. The main script accepts three options:

- `--nca` to run only the benchmarks for translation of the input regex into CA,
- `--dca` to run only the benchmarks for determinization of the CA,
- `--unanchored` to run benchmarks using regexes with `.*` as a prefix and suffix and with `s` flag.

Each of the individual benchmarks compiles the RE2 library with the corresponding flag to use only the desired part of the algorithm. It then runs it on a set of regexes. For each of the regexes, it prints out the regex, and the time it took to process it in milliseconds. After the last regex, it also prints the number of timeouts and the sum of the times.

The timeout of the individual benchmarks is set to 1 800 seconds. It can be changed using the `TIMEOUT_MS` constant. For the `--nca` option of the script, the constant has to be changed in the `constantsNondeterministicBenchmarks.py` file. For the `--dca` option of the script, the constant has to be changed in the `constantsDeterministicBenchmarks.py` file. Both files are located in the `/pythonScripts/helpers` folder.