BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2021
http://excel.fit.vutbr.cz

Beat Grep with Counters

Bc. Michal Horký*

Abstract
Regular expression (regex) 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 (regex) 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. The implementation within an existing library has the advantage of using its already implemented and optimized parts. We hope that using such parts helps our implementation to be even faster and outperforms some of the state-of-the-art matchers in matching regexes with counting operators while preserving the advantages of the original algorithm for other kinds of regexes.

Keywords: regular expression matching — bounded repetition — ReDoS — counting automata — RE2

Supplementary Material: *N/A*

*xhorky23@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Regular expressions (regexes) are widely supported by the most popular programming languages. About 30 – 40% of the Python, Java, and JavaScript software uses regular expression matching [1, 2, 3]. The usage of regexes includes searching, data validation, parsing, finding and replacing, data scraping, or syntax highlighting [4]. Regexes are using quantifiers to provide extended possibilities to describe repeated patterns. One of the quantifiers is the *counting operator*. The counting operator, also known as the operator of bounded repetition, can succinctly express repeated patterns. For example, the pattern `(qwe){3,6}` denotes all strings where `qwe` appears 3–6 times. As an analysis done in [4] suggest, the counting operator is often used in the regexes in practice.

The regex matching engines are based on automata theory; therefore, they mostly use algorithms based on *non-deterministic finite automata* (NFA) or *deterministic finite automata* (DFA). Probably the most implemented ones are *backtracking-based* search algorithms. These algorithms construct an NFA from the input regex and then simulates it on the input text. However, the backtracking can lead to *super-linear behavior* (SL behavior) when the complexity of the algorithm is polynomial or exponential in the input length. This behavior is vulnerable to the *regular expression denial of service* (*ReDoS*) attack. ReDoS is an attack that uses the super-linear behavior of regex matching to divert server resources from legitimate clients [2].

The DFA can be used either pre-computed (a static DFA simulation) or can be created on-the-fly (NFA-to-DFA simulation). The static DFA simulations have much lower worst-case complexity than the backtracking approach. More precisely, the matching can be linear to the input text length [5]. The major problem of this algorithm is the pre-computation of the DFA, which can suffer from the state explosion. This problem can cause significant performance issues when using the method in practice [4].

The NFA-to-DFA simulation, based on Thompson's algorithm [6], works directly with the NFA. In this way, it avoids the state explosion. The determinization is done on-the-fly. It always remembers only the current state. Then, every time it reads a new character from the input text, it computes the following DFA state, which replaces the current state. Even though this approach avoids the state explosion, the complexity of computation of the following DFA state is linear to the size of the NFA. It is because the DFA states (i.e., the sets of NFA states) could be large for highly non-deterministic NFAs. It can be partially solved by using a cache, in which are already visited states stored [4].

The exploding determinization is a problem for both static and NFA-to-DFA simulation. The frequent cause of it is the usage of the counting operator mentioned above. To eliminate this cause of the explosion, the paper by Turoňová et al. [4] proposes a novel succinct and fast deterministic machine called the *counting-set automaton* (CsA). It uses a special type of registers that can hold a set of bounded integers, called the *counting sets*, to succinctly express the counting operator. CsA also supports a limited selection of simple set operations, which, using appropriate data structures, can be implemented to run in *constant time* regardless of the size of the set. Thanks to these features, it can be used for the fast matching of regexes with the counting operator [4].

Matching based on CsAs is implemented within a C# prototype called *CA* [1]. The implementation is also experimentally evaluated in [4]. The experiments will be discussed in more detail later on; however, they show that other matchers can be just as fast or even faster than the C# implementation for non-counting heavy regexes. Our idea behind the implementation within RE2 is that such implementation can take advantage of overall RE2 speed and apply it to the CsAs based matching. We hope that in this way, we can make the matching of counting heavy regexes faster than the C# implementation. Our original intent was to implement the CsAs based matching into the grep. However, as described in Section 4, the RE2 is faster, so we implement the algorithm into the RE2.

We also present preliminary results where we show that our current steps of the implementation are faster than the C# implementation.

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

2. Counting Automata

This section is adopted from [4], where the CA is introduced. Counting automata (CAs) are a limited subclass of classical counter automata for regexes with counting. They are non-deterministic automata that can succinctly express regexes with counting. Before we can define the automaton itself, we must define the following notions.

A *counting algebra* is an effective Boolean algebra $\mathbb{C}$ associated with finite set C of counters. The counters have a lower and upper bound corresponding to the bounds of the counted repetition of the regexes. The bounds are $min_c \geq 0$ and $max_c > 0$, respectively, such that $min_c \leq max_c$.

Counter memories are the set of interpretations $\mathfrak{m}\colon C \to \mathbb{N}$ such that $\forall c \in C\colon 0 \leq \mathfrak{m}(c) \leq max_c$. Counter memories form the universe $\mathfrak{D}_\mathbb{C}$ of the effective Boolean algebra $\mathbb{C}$.

Combinations of basic predicates $\textsc{CanExit}_c$ and $\textsc{CanIncr}_c$ for $c \in C$ forms the set of predicates $\Psi_\mathbb{C}$ of the algebra $\mathbb{C}$. The semantic of these predicates is defined as follows:

$$\begin{aligned}\mathfrak{m} &\models \textsc{CanExit}_c \iff \mathfrak{m}(c) \geq min_c,\\ \mathfrak{m} &\models \textsc{CanIncr}_c \iff \mathfrak{m}(c) < max_c\end{aligned}$$

The counting automata itself is then defined as 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, $\Delta \subseteq Q \times \Psi_\mathbb{I} \times (C \to \mathscr{O}) \times Q$ is the (finite) transition relation, where $\mathscr{O} = \{\textsc{exit}, \textsc{incr}, \textsc{exit}1, \textsc{noop}\}$ is the set of counter operations, the component f of a transition $(p, \alpha, f, q) \in \Delta$ is its (counter) operator. f is often viewed as the set of indexed operations $\textsc{op}_c$, where $\textsc{op}$ denotes the operation assigned to the counter c, $f(c) = \textsc{op}$.

For each indexed operation $\textsc{op}_c$ a counter guard $\texttt{grd}(\textsc{op}_c)$ and a counter update $\texttt{upd}(\textsc{op})$ are defined as follows:

$$\begin{aligned}
\texttt{grd}(\textsc{noop}_c) &\overset{\text{def}}{=} \top_\mathbb{C} & \texttt{upd}(\textsc{noop}) &\overset{\text{def}}{=} \lambda n.n,\\
\texttt{grd}(\textsc{incr}_c) &\overset{\text{def}}{=} \textsc{CanIncr}_\mathbb{C} & \texttt{upd}(\textsc{incr}) &\overset{\text{def}}{=} \lambda n.n+1,\\
\texttt{grd}(\textsc{exit}_c) &\overset{\text{def}}{=} \textsc{CanExit}_\mathbb{C} & \texttt{upd}(\textsc{exit}) &\overset{\text{def}}{=} \lambda n.0,\\
\texttt{grd}(\textsc{exit}1_c) &\overset{\text{def}}{=} \textsc{CanExit}_\mathbb{C} & \texttt{upd}(\textsc{exit}1) &\overset{\text{def}}{=} \lambda n.1
\end{aligned}$$

The guard of the $\textsc{noop}$ operation is always enabled as the operation does not modify the counter. The $\textsc{incr}$

operation increments the counter, and it is enabled if the counter has not yet reached its upper bound. Guards of both EXIT and EXIT1 are enabled when the counter reaches its lower bound. The EXIT operation then resets the counter value to zero. The EXIT1 resets the counter value to one as it is the EXIT operation followed by the INCR operation.

On each transition, there can be more than one counter operation. In order to make the transition, all the guards of the operations on the transition must be enabled (formally, $\bigwedge_{\text{OP}_c \in f} \texttt{grd}(\text{OP}_c)$ has to be True). When all guards are enabled, and the transition is taken, all counters in the counter memory $\mathfrak{m}$ are updated by their corresponding operation.

The translation of regexes to CAs is done by generalized Antimirov's derivative construction [7]. The translation of the regex into the CA is described in more detail in [4]. Here, we present only the equations defining the conditional derivatives of a normalized regex, used during the translation. The equations are defined as follows:

$$\partial_\alpha(\varepsilon) \overset{\text{def}}{=} \emptyset \tag{1}$$

$$\partial_\alpha(\psi Z) \overset{\text{def}}{=} \begin{cases} \{\langle ID, Z\rangle\} & \text{if } \alpha \wedge \psi \text{ is satisfiable} \\ \emptyset & \text{otherwise} \end{cases} \tag{2}$$

$$\partial_\alpha((W \mid Y)Z) \overset{\text{def}}{=} \partial_\alpha(WZ) \cup \partial_\alpha(YZ) \tag{3}$$

$$\partial_\alpha(S*Z) \overset{\text{def}}{=} \partial_\alpha(S) \otimes \langle ID, S*Z\rangle\} \cup \partial_\alpha(Z) \tag{4}$$

$$\begin{aligned}\partial_\alpha(XZ) \overset{\text{def}}{=} &\ \partial_\alpha(S) \otimes \{\langle \text{INCR}_X, XZ\rangle\} \\ &\cup \{\langle \text{EXIT}_X, \varepsilon\rangle\} \otimes \partial_\alpha(Z)\end{aligned} \tag{5}$$

An example of the counting automaton for the regex $a\{1,3\}b\{5,9\}ab$ is in Figure 1a. We demonstrate how the transition is made on the transition from state q_0 to state q_1. The first step is to check if all guards of the counter operations on the transition are enabled. For the operation $\text{EXIT}_{a\{1,3\}}$ that means to check if the current value of the counter $a\{1,3\}$ is greater than or equal to the lower bound of the counting loop (which is one for this particular loop). For the operation $\text{INCR}_{b\{5,9\}}$ that means to check if the current value of the counter $b\{5,9\}$ is less than the upper bound of the counting loop (which is nine for this particular loop). If both guards are enabled, the automaton can make the transition. As part of it, it also updates the corresponding counter values, specifically the $\text{EXIT}_{a\{1,3\}}$ operation resets the value of the counter $a\{1,3\}$ to zero, and the $\text{INCR}_{b\{5,9\}}$ operation increments the current value of the counter $b\{5,9\}$ by one. Note that all counters are initially set to zero.

3. Counting-Set Automata

Counting automata presented in the previous section are non-deterministic. Before the automaton can be used for matching, it must be firstly determinized. This section is adopted from [4], where the CsA is introduced.

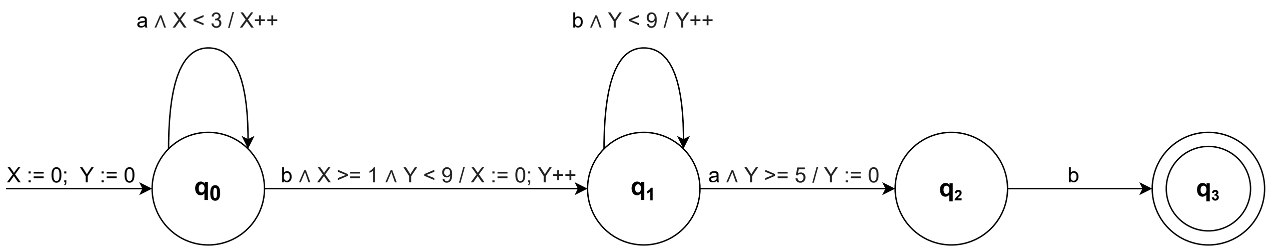
When the NFA and the DFA are used for regexes with counting operators, the NFA is created with counter memories being an explicit part of control states. The NFA is then determinized by the textbook subset construction. This approach is prone to the state explosion since the explosion can occur already in the creation of the NFA because of the memories being part of the control states. The explosion can be even worse in the determinization process since the subset construction is exponential in the size of the NFA.

The *counting-set automata* (*CsAs*), which are directly created by the determinization of CAs, are the solution to this problem. The states of the CsA are essentially states of the corresponding DFA without the counter memories. To replace the counter memories from the DFA states, the CsA has special registers that can hold sets of integers. These registers are used at runtime to compute the right values of the counters. Such construction avoids the state explosion mentioned above.

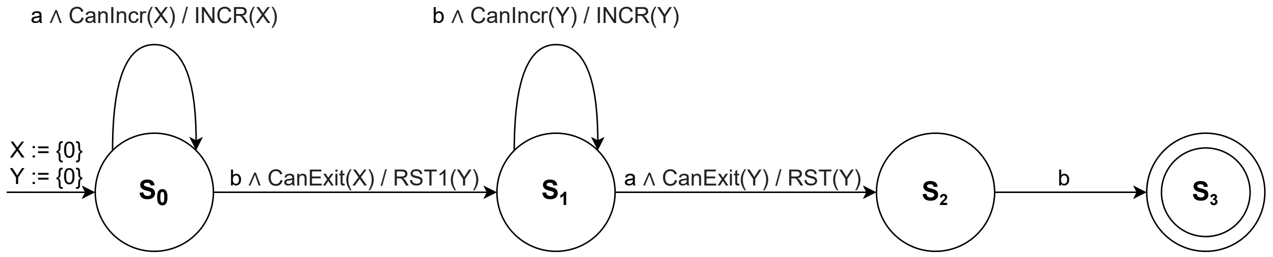
To allow manipulation with pairs of predicates from the input algebra $\mathbb{I}$ and the counting-set algebra $\mathbb{S}$, the notion of a combined Boolean algebra $\mathbb{I} \times \mathbb{S}$ is used. We assume that predicates in $\Psi_{\mathbb{I}\times\mathbb{S}}$ have a form $\alpha \wedge \beta$ 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, meaning that the value of a counter c is a *finite set*. A counter is then called a *counting set*. Let $\mathscr{P}_{fin}(X)$ denote the powerset of X, which is restricted to finite non-empty sets. Then a function $\mathfrak{s}\colon C \to \mathscr{P}_{fin}(\mathbb{N})$, such that $\forall c \in C\colon Max(\mathfrak{s}(c)) < max_c$ is called a *counting set memory* for C.

An effective Boolean algebra $\mathbb{S}_C$ called the *counting-set algebra over* C is an algebra formed by counting-set predicates over C. 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:



(a) 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), incrementation of the counter value is denoted by ++ (e.g., X++). The character class (in this example, only a single letter) is written as first on the transition; then there are guards of the corresponding counter operations, and, after / symbol, there are the operations themselves. The guards and the operations are shown only if there are any. Note that X := 0 and Y := 0 represents the EXIT operation of the corresponding counter as it resets the value of the counter to zero, and X++ and Y++ represents the INCR operation of the corresponding counter as it increments the value of the counter by one.



(b) Counting-set automaton for the counting automaton. 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.

Figure 1. Counting automaton created by the derivative based construction from the regex $a\{1,3\}b\{5,9\}ab$ and counting-set automaton obtained as a result of the determinization of the counting automaton. Note that the counting automaton uses an intuitive notation and the counting-set automaton follows the formal notation.

$$\mathfrak{s} \models \textsc{CanExit}_c \Leftrightarrow Max(\mathfrak{s}(c)) \geq min_c,$$
$$\mathfrak{s} \models \textsc{CanIncr}_c \Leftrightarrow Min(\mathfrak{s}(c)) < max_c$$

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.

A counting-set automaton (CsA) itself is then defined as a tuple $A = (\mathbb{I}, C, Q, F, \Delta)$. The individual components are 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, and $\Delta \subseteq Q \times \Psi_{\mathbb{I}\times\mathbb{S}} \times (C \to \mathscr{P}(\mathscr{O})) \times Q$ 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 is the *counting-set operator*, where $\mathscr{O}$ denotes the set $\{\textsc{Incr}, \textsc{Noop}, \textsc{Rst}, \textsc{Rst}1\}$ of *counting-set operations*. The counting-set operations are basically counter operations, just lifted to sets. In order to distinguish them, the counting-set operations are written capitalized. Also, the lifted counter operations EXIT and EXIT1 are written as RST and RST1, respectively. That is because they are also used for initialization when entering the loop, and therefore, the usage is different from the counter operations EXIT and EXIT1. The counting-set operator assigns sets of counting-set operations to counters; these sets are called *combined (counting-set) operations*.

The CsA A is *deterministic* iff the following holds for every two transitions $p(\psi_1, f_1)q_1$ and $p(\psi_2, f_2)q_2$ 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 $\textsc{op}_c \in \mathscr{O}$ is the set transformer $\mathtt{upd}(\textsc{op}_c)$, which is defined as follows:

$$\begin{aligned}
\mathtt{upd}(\textsc{incr}_c) &\overset{\text{def}}{=} \lambda S.\{n+1 \mid n \in S \wedge n < max_c\}\\
\mathtt{upd}(\textsc{rst}_c) &\overset{\text{def}}{=} \lambda S.\{0\},\\
\mathtt{upd}(\textsc{noop}_c) &\overset{\text{def}}{=} \lambda S.S\\
\mathtt{upd}(\textsc{rst}1) &\overset{\text{def}}{=} \lambda S.\{1\}
\end{aligned}$$

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

$$\mathtt{f} \overset{\text{def}}{=} \lambda c. \begin{cases} \bigcup_{\text{OP} \in f(c)} \mathtt{upd}(\text{OP}_c)(\mathfrak{s}(c)) & \text{if } f(c) \neq \emptyset \\ \{0\} & \text{if } f(c) = \emptyset \end{cases} \tag{6}$$

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.

In terms of the guards, it is necessary to distinguish cases such as $\neg\text{CANEXIT}_c \wedge \text{CANINCR}_c$, $\text{CANEXIT}_c \wedge \neg\text{CANINCR}_c$, or $\text{CANEXIT}_c \wedge \text{CANINCR}_c$. 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.

The whole determinization process is described in [4]. Here, we define only basic notions and illustrate the determinization on an example.

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

1. there is a transition to q with either INCR_c or EXIT1_c,
2. there is a transition to q from a state in $\sigma(c)$ with 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 EXIT_c takes place.

Let R be a set of CA states that forms a CsA state. The set of minterms $\Gamma_{R,\alpha}$ of the set of counter guards on the $\alpha-$transitions originating in R is defined as follows:

$$\Gamma_{R,\alpha} \overset{\text{def}}{=} \mathit{Minterms}(\{\mathtt{grd}(\text{OP}_c) \mid (r,\alpha,f,s) \in \Delta, \quad r \in R \wedge c \in \sigma(r), \text{OP}_c \in f\}) \tag{7}$$

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:

$$\Delta_{R,\alpha,\beta} \overset{\text{def}}{=} \{(r,\alpha,f,s) \in \Delta \mid r \in R, \mathit{Sat}(\varphi_f \wedge \beta)\} \tag{8}$$

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. The counting-set operation for the counter c obtained from the CA transition $\tau = p(\alpha, f)q$ is defined by the following equations:

$$\text{NOOP} \quad \text{if } f(c) = \text{NOOP} \wedge p \in \sigma(c) \tag{9}$$
$$\text{INCR} \quad \text{if } f(c) = \text{INCR} \wedge p \in \sigma(c) \tag{10}$$
$$\text{RST} \quad \text{if } f(c) = \text{NOOP} \wedge p \notin \sigma(c) \tag{11}$$
$$\text{RST1} \quad \text{if } f(c) = \text{INCR} \wedge p \notin \sigma(c) \tag{12}$$
$$\text{RST} \quad \text{if } f(c) = \text{EXIT} \tag{13}$$
$$\text{RST1} \quad \text{if } f(c) = \text{EXIT1} \tag{14}$$

The counting-set automaton created by determinization of the counting automaton for the regex $a\{1,3\}b\{5,9\}ab$ is shown in Figure 1b. We demonstrate the determinization process 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: CANEXIT_X and $\neg\text{CANEXIT}_X$. Since there is CANEXIT_X on the original transition, the element $\neg\text{CANEXIT}_X$ of the gamma set can not be used as $\text{CANEXIT}_X \wedge \neg\text{CANEXIT}_X$ is not satisfiable. We now have the character class (a single letter in our example) b, the guard CANEXIT_X, and we need to get the operation. The EXIT_X operation is irrelevant since X is not in the scope of the state q_1. For the operation INCR_y, we get the RST1 operation by using Equation 12. From these components, we get the transition $S_0(b \wedge \text{CANEXIT}_X, \{\text{RST1}_Y\})S_1$.

4. C# Implementation

The paper by Turoňová et al. [4] also introduces a C# implementation of CsA-based matching called CA. The speed of the implementation is also experimentally evaluated and compared to the state-of-the-art matchers, including RE2 and grep. In Figure 2, there is a comparison of CA and RE2, and in Figure 3, there is a comparison of CA and grep. It is clear that thanks to many optimizations, the RE2 wins more often. However, as stated in the paper, there are 287 benchmarks where the performance of RE2 significantly deteriorates, and the CA is faster. Also, in 89 benchmarks,

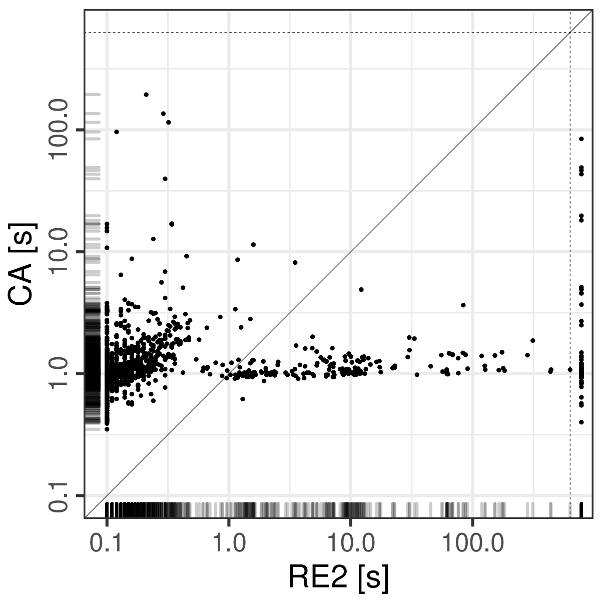


Figure 2. The comparison of running times of CA and RE2 on the benchmark set from [4] (CA wins: 287/1,789).

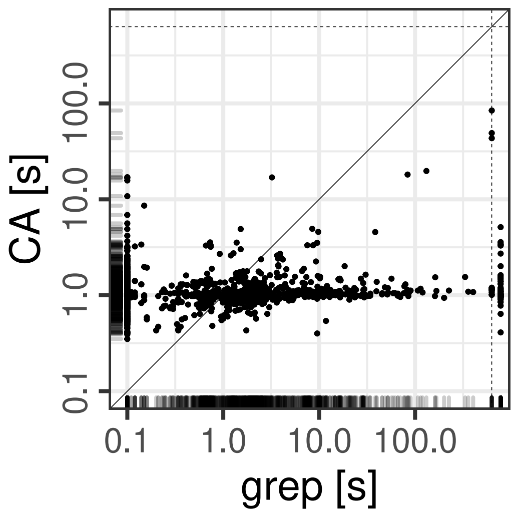


Figure 3. The comparison of running times of CA and grep on the benchmark set from [4] (CA wins: 862/1,425).

the matching time of RE2 is bigger than 10s, which can be considered as a successful ReDoS attack.

The RE2 was evaluated as the fastest matcher overall in the paper by Turoňová et al. [4]. It is also clear that the RE2 CA beats grep more often than it beats the RE2. We also include grep in this comparison, since comparing with grep was the original motivation of this work.

These benchmarks show that RE2 is faster than CA on the regexes where using the counting-set automata does not bring such a speedup. However, there are regexes where usage of the counting-set automata has a significant impact, and the RE2 is much slower despite all the optimizations.

That is the main observation that led us to the idea of implementing the counting-set automata matching within the RE2 matcher. We want to combine the overall speed of RE2 (which is archived by very optimized RE2 implementation with a combination of the speed of the C++ language) with the counting-set automata technique which speeds up matching of regexes with counting operator significantly, and get the best of both matchers in one.

5. Our Implementation

As stated before, we are implementing the counting-set automata based matching within the RE2 matcher. The implementation follows the derivative-based CA construction and the determinization from the paper by Turoňová et al. [4]. The implementation is divided into three main steps:

1. translation of the regex to the CA by partial derivative construction,
2. creating CsA as the result of the CA determinization done by the algorithm from the paper,
3. the matching itself based on the CsA obtained in the previous step.

In the first step, the regex is firstly parsed by the already implemented functions of RE2. The internal representation of the regex is then used to perform the derivative-based construction of the CA. This part of the algorithm implements Equations 1–5. However, these equations do not include a definition for some of the regex operators—specifically the plus and the question mark operator. We decided not to introduce new equations for these operators in order to avoid possible negative consequences of such action. Instead, we decided to convert these operators to other operators for which the equations are already defined. The plus operator is converted to the star operator. For the plus operator, the subexpression must be repeated at least once, while the star operator does not enforce any repetition. Therefore, we extract the subexpression and concatenate it with the original repetition while changing the plus operator to the star operator. For example, the regex $(abc)+$ will be converted to the regex $abc(abc)*$, which has the same meaning, the expression can be repeated, but there must be at least one occurrence of the expression. The question mark means zero or one occurrence of the pattern. We convert this operator to the alternation, where one alternative is epsilon (representing zero occurrences), and the other alternative is the original expression (representing one occurrence). For example, the regex $a?$ will be converted to the regex $\varepsilon|a$.

The conversion of the plus operator is done as part of the normalization of the internal representation of the regex (more information about the normalization of the regex can be found in [4]), which is done before the computation of partial derivatives. The conversion of the question mark operator is done when computing the derivatives.

In the second step, the algorithm first computes the state scopes. As part of the computation, the algorithm also updates the counter operations, specifically the NOOP operations. When the NOOP operation is processed in the derivative construction, there is no information about the corresponding counter. Also, all the counters must have an implicit NOOP operation on every transition where the counter is not used. Otherwise, the set of minterms $\Gamma_{R,\alpha}$, defined in Equation 7, would not be computed right since it works with the counter associated with the operation. However, in this step, the algorithm has enough information about the counters to add the corresponding counter to NOOP operations and also add implicit NOOP operation to all transitions where the counter is not used.

The algorithm then starts to explore states of the CA, starting in the initial state. For each state, it explores all outgoing transitions, for which it computes the set of minterms $\Gamma_{R,\alpha}$. It also computes the counting-set operator f; however, in this step, it does not yet reflect the scopes of the states.

The counting-set automaton is currently precomputed. It allows us simpler verification of the results of the computation than the on-the-fly determinization. However, when the determinization step is completed, we will try to implement the on-the-fly determinization. We are currently taking it into account while we are writing the corresponding functions.

The third step, the matching itself, has not yet been implemented. We are currently working on this part of the algorithm, and we hope that it will be completed in the upcoming months.

6. Preliminary Results

Since the algorithm is not fully completed, we can evaluate and compare only the translation of the regex into counting automata and determinization of the counting automata. We compare our implementation with CA matcher written in C. We tested the implementations only on more complex regexes. We have chosen 37 examples from the benchmark of [4] that were hard to determinize for the C# implementation.

The result in Table 1 shows that our implementation is significantly faster for both translations of the regex into counting automata and determinization of the counting automata. The determinization part of our algorithm takes a long time on some regexes on which the CA implementation time outs. To stretch out the difference, we also provide the result of our algorithm on regexes on which the CA does not time outs. Also, the number of regexes for which the algorithm timeouts is two-thirds lower, so we hope that our implementation will also be usable for more regexes than the CA. We note that these are preliminary results of our first version of the implementation, which we are still developing.

	CA		Our implementation		
	NCA	CsA	NCA	CsA	CsA without timeouts
mean	10 966	73 433	8	52 842	3 774
median	254	11 338	3	879	693
timeouts	0	14	0	5	N/A

Table 1. Experimental evaluation of our implementation and CA (time 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. CsA without timeouts denotes the time of the determinization, which was run only on regexes on which the CA implementation does not time outs. The timeout was set to 18 000 seconds.

7. Conclusion

In this paper, we presented an implementation of the counting-set automata based regular expression matching. We are implementing our algorithm within the state-of-the-art matcher RE2.

Our preliminary results show that the combination of fast RE2 matcher with the CsA based matching can be capable of outperforming other state-of-the-art matchers.

The steps that are ahead of us are the finalization of determinization and the matching itself. Also, we will try to implement the on-the-fly determinization, which could speed up the matching even more.

Acknowledgements

I would like to thank my supervisor Mgr. Lukáš Holík, Ph.D. for his help and advices, and Ing. Lenka Turoňová for the benchmark dataset.

References

[1] James C. Davis. Rethinking regex engines to address redos. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering*

Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, page 1256–1258, New York, NY, USA, 2019. Association for Computing Machinery.

[2] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 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*, ESEC/FSE 2018, page 246–256, New York, NY, USA, 2018. Association for Computing Machinery.

[3] Carl Chapman and Kathryn T. Stolee. Exploring regular expression usage and context in python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 282–293, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. Regex matching with counting-set automata. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[5] Michael Sipser. Introduction to the theory of computation. *SIGACT News*, 27(1):27–29, March 1996.

[6] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[7] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291 – 319, 1996.