Fast Matching of Regular Patterns with Synchronizing Counting

Lukáš Holík, Juraj Síč(✉), Lenka Turoňová, and Tomáš Vojnar

Brno University of Technology, Brno, Czech Republic
{holik,sicjuraj,ituronova,vojnar}@fit.vut.cz

Abstract. Fast matching of regular expressions with *bounded repetition*, aka *counting*, such as `(ab){50,100}`, i.e., matching linear in the length of the text and independent of the repetition bounds, has been an open problem for at least two decades. We show that, for a wide class of regular expressions with counting, which we call *synchronizing*, fast matching is possible. We empirically show that the class covers nearly all counting used in usual applications of regex matching. This complexity result is based on an improvement and analysis of a recent matching algorithm that compiles regexes to deterministic counting-set automata (automata with registers that hold sets of numbers).

1 Introduction

Fast matching of regular expressions with *bounded repetition*, aka *counting*, has been an open problem for at least two decades (cf., e.g., [33]). The time complexity of the standard matching algorithms run on a regex such as `.*a.{100}` is, at best, dominated by the *length of the text multiplied by the repetition bounds*. This makes matching prone to unacceptable slowdowns since the length of the text as well as the repetition bounds are often large. In this paper, we provide a theoretical basis for matching of bounded repetition with a much more reliable performance. We show that a large and practical class of regexes with counting theoretically allows **fast matching**—in time **independent of the counter bounds** and **linear in the length of the text**.

The problem also has a strong practical motivation. Regex matching is used for searching, data validation, detection of information leakage, parsing, replacing, data scraping, syntax highlighting, etc. It is natively supported in most programming languages [6], and ubiquitous (used in 30–40 % of Java, JavaScript, and Python software [7,39,8,5]). Efficiency and predictability of regex matching is important. An extreme run-time of matching can have serious consequences, such as a failed input validation against injection attacks [41] and events like the outage of Cloudflare services [18]. Regexes vulnerabilities are also a doorway for the *ReDoS (regular expression denial of service) attack*, in which the attacker crafts a text to overwhelm a matcher (as, e.g., in the case of the outage of StackOverflow [13] or the websites exposed due to their use of the popular Express.js framework [3]). ReDoS has been widely recognized as a common and serious threat [7,9,11], with counting in regexes begin especially dangerous [37].

© The Author(s) 2023
O. Kupferman and P. Sobocinski (Eds.): FoSSaCS 2023, LNCS 13992, pp. 392–412, 2023.
https://doi.org/10.1007/978-3-031-30829-1_19

Matching algorithms and complexity. The potential instability of the pattern matchers is in line with the worst-case complexity of the matching algorithms. The most widely used approach to matching is backtracking (used, e.g., in standard matchers of .NET, Python, Perl, PHP, Java, JavaScript, Ruby) for its simplicity and ease of implementation of advanced features such as back-references or look-arounds. It is, however, at worst exponential to the length of the matched text and prone to ReDoS. Even though this can be improved, for instance by memoization [11], the fastest matchers used in performance critical applications all use automata-based algorithms instead of backtracking. The basis of these approaches is Thompson's algorithm [35] (also referred to as *online NFA-simulation*). Together with many optimizations, it is implemented in Intel's Hyperscan [40]. When combined with caching, it becomes the on-the-fly subset construction of a DFA, also called *online DFA-simulation* (implemented in RE2 from Google, GNU grep, SRM, or the standard matcher of Rust [17,19,30,12]). Without counting, the major factor in the worst-case complexity is $O(nm^2)$, with n being the length of the text and m the size of the number of character occurrences in the regex (m is smaller than size of the regex, the length of string defining it). We say that the *character cost*, i.e., the cost of extending the text with one character, is m^2. This is the cost of iterating through transitions of an NFA with $O(m)$ states and $O(m^2)$ transitions compiled from the regex by some classical construction [2,16,24].

Extending the syntax of regexes with *bounded quantifiers* (or *counters*), such as `(ab){50,100}`, increases the character complexity dramatically. Given k counters with the maximum bound ℓ, the number of NFA states rises to $O(m\ell^k)$, the number of transitions as well as the character cost to $O((m\ell^k)^2)$. For instance, the minimal DFA for `.*a.{k}` (i.e., a appears k characters from the end) has more than 2^k states. Moreover, note that, since k is written as a decadic numeral, its value is exponential in the size of the regex. This makes matching with already moderately high k prone to significant slowdowns and ReDoS vulnerabilities with virtually every mainstream matcher (see [36,37]). At the same time, repetition bounds easily reach thousands, in extreme tens of millions (in real-life XML [4]). Writing a dangerous counting expression is easy and it is hard to identify. Security-critical solutions may be vulnerable to counting-related ReDoS [37] despite an extra effort spent in regex design and testing, hence developers sometimes avoid counting, use workarounds and restrict functionality.

The problem of matching with bounded repetition has been addressed from the theoretical as well as from the practical perspective by a number of authors [15,4,22,26,31,20,25,36]. From these, the recent work [36] is the only one offering fast matching for a practically significant class of regexes. The algorithm of [36] compiles a regex with counting to a non-deterministic *counting automaton (CA)*, an automaton with counters that can be incremented, reset, and compared with a constant. The crux of the problem is then to convert the CA to a succinct deterministic machine that could be simulated fast in matching. The work [36] achieves this by determinizing the CA into a *counting-set automaton (CSA)*, an automaton with registers that hold *sets* of numbers. Its size is independent of the counter bounds and it updates the sets by a handful of operations that are all constant time, regardless the size of the sets. However, regexes outside the supported class do appear, the class has no syntactic characterization, and it is hard to recognize (as demonstrated also by an incorrect proposal of a syntactic

class in [36] itself). For instance, `.*a{5}` or `(ab){5}` are handled, but `.*(aa){5}` or `.*(ab){5}` are not (the requirement is technical, see Section 4).

Our contribution. In this paper, we

1. **generalize the algorithm of [36] to extend the class of handled regexes and**
2. **derive a useful syntactic characterization of the extended class.**

The derived class is characterized by *flat counting* (counting operators are not nested) where repetitions of each counted expression R are *synchronizing* (a word from R^n cannot have a prefix from R^{n+1}). It is the first clearly delimited practical class of regexes with counting that allows fast matching. It includes the easily recognizable and frequent case where every word in R has exactly one occurrence of a *marker*, a letter or a word from a finite set of markers that unambiguously identifies each occurrence of R (note that even this simple class was not handled by any previous fast algorithms, including [36]). In a our experiment with a large set of regexes from various sources, 99.6 % of non-trivial flat counting was synchronizing and 99.2 % was letter-marked.

To obtain the results (1) and (2) above, **we first modify the determinization of [36] to include the entire class of regexes with flat counting**. In a nutshell, this is achieved by two changes: (i) We allow copying and uniting of sets stored in registers, and (ii) in the determinization, we index counters of the CA by its states to handle CA in which nondeterministic runs that reach different states reach different counter values.

These modifications come with the main technical challenge that we solve in this paper: copying and uniting sets is not constant-time but linear to the size of the sets. This would make the character cost linear in the counter bound ℓ again. To remove the dependency on the counter bounds, we augment the determinization by optimizations that avoid the copying and uniting. First, to alleviate the cost of uniting, we store intersections of sets stored in registers in new shared registers, so that the intersection does not contribute to the cost of uniting the registers. Then, to increase the impact of intersection sharing, we synchronize register updates in order to make their intersections larger. We then show that if the CSA *does not replicate registers*, i.e, each register can in a transition appear on the right-hand side of only one register assignment, then it never copies registers and the cost of unions can be amortised. Finally, **we define the class of regexes with *synchronizing counting* for which the optimized CsA do not replicate counters so their simulation in matching is fast.**

Related work. In the context of regex matching, counting automata were used in several forms under several names (e.g. [20,36,4,15,31,32,33,14,23]). Besides [36] discussed above, other solutions to matching of counting regexes [15,4,22,26,31,20,25] handle small classes of regexes or do not allow matching linear in the text size and independent of counter bounds. The work [20] proposes a CA-to-CA determinization producing smaller automata than the explicit CA determinization for the limited class of monadic regexes, covered by letter-marked counting, and the size of their deterministic automata is still dependent on the counter bounds. The work [4] uses a notion of automata with counters of [15]. It focuses mostly on deterministic regexes, a class much smaller than regexes with synchronizing counting, and proposes a matching algorithm still dependent on the counter bounds. The paper [25] proposes an algorithm that takes time at

worst quadratic to the length of the text. Extended FA (XFA) of [31,32] augment NFA with a scratch memory of bits that can represent counters, and their determinization is exponential in counter bounds already for regexes such as `.*a.`$\{k\}$. The *counter-1-unambiguous* regexes of [22,23] can be directly compiled into deterministic automata called FACs, similar to our CA, independent of counter bounds, but the class is limited, excluding e.g., `.*a.`$\{k\}$.

2 Preliminaries

We use $\mathbb{N}$ to denote the natural numbers including 0. For a set S, $\mathcal{P}(S)$ denotes its powerset and $\mathcal{P}_{\text{fin}}(S)$ is the set of all *finite* subsets of S.

A *first order language (f.o.l.)* $\Gamma = (F, P)$ consists of a set of *function symbols* F and a set of *predicate symbols* P. An *interpretation* $\mathbb{I}$ of Γ with a *domain* $D_{\mathbb{I}}$ assigns a function $f^{\mathbb{I}} : D_{\mathbb{I}}^n \to D_{\mathbb{I}}$ to each n-ary $f \in F$ and a function $p^{\mathbb{I}} : D_{\mathbb{I}}^n \to \{0,1\}$ to each n-ary $p \in P$. An *assignment* of a set of variables X in $\mathbb{I}$ is a total function $\nu : X \to D_{\mathbb{I}}$. The set of *terms* $\mathsf{Terms}_{\Gamma,X}$ and the set $\mathsf{QFF}_{\Gamma,X}$ of *quantifier free formulae* (boolean combinations of atomic formulae) over Γ and X, as well as the interpretation of a term, $t^{\mathbb{I}}(\nu)$, and a formula, $\varphi^{\mathbb{I}}(\nu)$, are defined as usual. We denote by $\nu \models_{\mathbb{I}} \varphi$ that the formula φ is *satisfied* (interpreted as true) by the assignment ν. It is then *satisfiable*. We drop the sub/superscript $\mathbb{I}$ when it is clear from the context. We write $\varphi[x]$ and $t[x]$ to denote a unary formula φ or term t, respectively, with the free variable x, and we may also abuse this notation to denote the term/formula with its only free variable replaced by x. We write $t^{\mathbb{I}}(k)$ and $\varphi^{\mathbb{I}}(k)$ to denote the values $t^{\mathbb{I}}(\{x \mapsto k\})$ and $\varphi^{\mathbb{I}}(\{x \mapsto k\})$. For a set of formulae $\Psi = \{\psi_1, \ldots, \psi_n\}$, the set *Minterms*$(\Psi)$ consists of all *minterms* of Ψ, satisfiable conjunctions $\varphi_1 \wedge \cdots \wedge \varphi_n$ where for each $i : 1 \leq i \leq n$, φ_i is ψ_i or $\neg\psi_i$.

We fix a finite *alphabet* Σ of *symbols/letters* for the rest of the paper. Words are sequences of letters, with the *empty word* ε. The *concatenation* of words u and v is denoted $u \cdot v$, uv for short. A set of words over Σ is a *language*, the concatenation of languages is $L \cdot L' = \{u \cdot v \mid u \in L \wedge v \in L'\}$, LL' for short. *Bounded iteration* x^i, $i \in \mathbb{N}$, of a word or a language x is defined by $x^0 = \varepsilon$ for a word, $x^0 = \{\varepsilon\}$ for a language, and $x^{i+1} = x^i \cdot x$. Then $x^* = \bigcup_{i \in \mathbb{N}} x^i$. We consider a usual basic syntax of *regular expressions (regexes)*, generated by the grammar $R ::= \varepsilon \mid \texttt{a} \mid \texttt{(}R\texttt{)} \mid RR \mid R\texttt{|}R \mid R\texttt{*} \mid R\{m,n\}$ where $m \in \mathbb{N}$, $n \in \mathbb{N} \cup \infty$, $0 \leq m$, $0 < n$, $m \leq n$, and $\texttt{a} \in \Sigma$. We use $R\{m\}$ for $R\{m,m\}$. Regexes containing a sub-expression with the *counter* $R\{m,n\}$ or $R\{m\}$ are called *counting regexes* and m,n are *counter bounds*. We denote by $\max_R$ the maximum integer occurring in the counter bounds of regex R and we denote the number of counters by cnt_R. A regex with *flat counting* does not have nested counting, that is, in a sub-regex $S\{m,n\}$, S cannot contain counting. The *language* of a regex R is constructed inductively to the structure: $L(\varepsilon) = \{\varepsilon\}$, $L(\texttt{a}) = \{a\}$ for $a \in \Sigma$, $L(RR') = L(R) \cdot L(R')$, $L(R\texttt{*}) = L(R)^*$, $L(R\texttt{|}R') = L(R) \cup L(R')$, and $L(R\{m,n\}) = \bigcup_{m \leq i \leq n} L(R)^i$. We understand $|R|$ simply as the length of the defining string, e.g. $|\texttt{(ab)}\{\texttt{10}\}| = 8$. We define $\sharp R$ as the number of character occurrences in R, formally, $\sharp a = 1$ for $a \in \Sigma$, $\sharp\varepsilon = 0$, $\sharp\texttt{(}R\texttt{)} = \sharp R\{\texttt{m,n}\} = \sharp R$, and $\sharp R \cdot S = \sharp R\texttt{|}S = \sharp R + \sharp S$.

A *(nondeterministic) automaton (NA)* is a tuple $A = (Q, \Delta, I, F)$ where Q is a set of *states*, Δ is a set of *transitions* of the form $q\text{-}\{a\}\!\!\to r$ with $q, r \in Q$ and $a \in \Sigma$, $I \subseteq Q$ is the

set of *initial states*, and $F \subseteq Q$ is the set of *final states*. A run of A over a word $w = a_1 \dots a_n$ from state p_0 to p_n, $n \geq 0$ is a sequence of transitions $p_0 \text{-}\{a_1\}\!\!\rightarrow p_1$, $p_1 \text{-}\{a_2\}\!\!\rightarrow p_2$, $\dots$, $p_{n-1} \text{-}\{a_n\}\!\!\rightarrow p_n$ from Δ. The empty sequence is a run with $p_0 = p_n$ over ε. The run is *accepting* if $p_0 \in I$ and $p_n \in F$, and the language $L(A)$ of A is the set of all words for which A has an accepting run. A state q is *reachable* if there is a run from I to it. The *size* of the NA, $|A|$, is defined as the number of its states plus the number of its transitions. The automaton is *deterministic (DA)* iff $|I| = 1$ and for every state q and symbol a, Δ has at most one transition $q \text{-}\{a\}\!\!\rightarrow r$. The *subset construction* transforms the NA to the DA with the same language $\mathtt{DA}(A) = (Q^{\mathbb{0}}, \Delta^{\mathbb{0}}, I^{\mathbb{0}}, F^{\mathbb{0}})$ where $Q^{\mathbb{0}} \subseteq \mathcal{P}(Q)$ and $\Delta^{\mathbb{0}}$ are the smallest sets of states and transitions satisfying $I^{\mathbb{0}} = \{I\}$, $\Delta^{\mathbb{0}}$ has for each $a \in \Sigma$ and each $S \in Q^{\mathbb{0}}$ the transition $S \text{-}\{a\}\!\!\rightarrow \{s' \mid s \in S \wedge s \text{-}\{a\}\!\!\rightarrow s' \in \Delta\}$, and $F^{\mathbb{0}} = \{S \in Q^{\mathbb{0}} \mid S \cap F \neq \emptyset\}$. When the set of states Q is finite, we talk about (deterministic) *finite state* automata (NFA, DFA).[1]

This paper is concerned with the problem of fast *pattern matching*, basically a membership test: given a regex R and a text w, decide whether $w \in L(R)$. While w may be very long, R is normally small, hence the dependence on $|w|$ is the major factor in the complexity. The offline DFA simulation takes time linear in $|w|$. It (1) compiles R into an NFA $\mathtt{NFA}(R)$ (2) determinizes it, and (3) follows the DFA run over w (aka *simulates* the DFA on w), all in time and space $\Theta(2^{|\mathtt{NFA}(R)|} + |w|)$. The cost of determinization, exponential in $|\mathtt{NFA}(R)|$, is however too impractical. Modern matchers such as Grep or RE2 [19,17] therefore use the techniques of online DFA simulation, where only the part of the DFA used for processing w is constructed. It reduces the complexity to $O(\min(2^{|\mathtt{NFA}(R)|} + |w|, |w| \cdot |\mathtt{NFA}(R)|))$ (the first operand of min is the explicit determinization in case the entire DFA is constructed, plus the cost of DFA-simulation; the second operand is the cost of the online-DFA simulation, coming from that every step may incur construction of a new DFA state and transition in time $O(|\mathtt{NFA}(R)|)$). For counting regexes, the factor $|\mathtt{NFA}(R)|$ depends linearly (or more if counting is nested) on $\max_R$ and thus exponentially on $|R|$. This makes counting very problematic in practice [36,37,33]. We will present a matching algorithm which is *fast* for a specific class of regexes, meaning that its run-time is still linear in $|w|$ but is independent of $\max_R$.

3 Counting Automata

We use a rephrased definition of counting automata and counting-set automata of [36]. We will present them as a special case of a generic notion of automata with registers.

Definition 1 (Automata with registers). *An* automaton with registers *(RA) operated through an f.o.l.* Γ *under an interpretation* $\mathbb{I}$ *is a tuple* $A = (X, Q, \Delta, I, F)$ *where* X *is a set of variables called* registers*;* Q *is a finite set of* states*;* Δ *is a finite set of* transitions *of the form* $q \text{-}\{a, \varphi, u\}\!\!\rightarrow p$ *where* $p, q \in Q$*,* $a \in \Sigma$*,* $u : X \rightarrow \mathsf{Terms}_{\Gamma,X}$ *is an* update*, and* $\varphi \in \mathsf{QFF}_{\Gamma,X}$ *is a* guard*;* I *is a set of* initial configurations*, where a* configuration *is a pair of the form* $(q, \mathfrak{m})$ *where* $q \in Q$ *and* $\mathfrak{m} : X \rightarrow D_{\mathbb{I}}$ *is a register assignment called a* memory*; and* $F : Q \rightarrow \mathsf{QFF}_{\Gamma,X}$ *is a* final condition assignment.

[1] We do not require finiteness in the basic definition in order to avoid artificial restrictions of the notions of automata with registers/counters/counting sets defined later.

The language of A, $L(A)$, is defined as the language of its configuration automaton $\mathtt{Conf}(A)$. *States of* $\mathtt{Conf}(A)$ *are* configurations *of A that are reachable. I is the set of initial states of* $\mathtt{Conf}(A)$. *It has a transition* $(q,\mathfrak{m})\text{-}\{a\}\text{→}(q',\mathfrak{m}')$ *iff* $(q,\mathfrak{m})$ *is reachable and A has a transition* $\delta = q\text{-}\{a,\varphi,u\}\text{→}q' \in \Delta$ *such that* $(q',\mathfrak{m}')$ *is the* image *of* $(q,\mathfrak{m})$ *under* δ, *denoted* $(q',\mathfrak{m}') = \delta(q,\mathfrak{m})$, *meaning that (1)* δ *is* enabled *in* $(q,\mathfrak{m})$, $\mathfrak{m} \models \varphi$, *and (2)* $\mathfrak{m}' = u(\mathfrak{m})$, *i.e.* $\mathfrak{m}'(x) = u(x)^{\mathbb{I}}(\mathfrak{m})$ *for each* $x \in X$. *We let* $\delta(C) = \{\delta(c) \mid c \in C\}$ *for a set of configurations C. A configuration* $(q,\mathfrak{m})$ *is a final if* $\mathfrak{m} \models F(q)$. *By* runs of *A we mean runs of* $\mathtt{Conf}(A)$. *The RA A is* deterministic *if* $\mathtt{Conf}(A)$ *is deterministic. The size of the RA is* $|A| = |Q| + \sum_{\delta\in\Delta}|\delta|$ *where* $|\delta|$ *is the sum of the sizes of the update and the guard.*

Definition 2 (Counting automata). *A* counting automaton *(CA) is an automaton with registers, called* counters, *operated through the* counting language $\Gamma_{\mathtt{cnt}}$ *that contains the unary increment function, denoted* $x+1$, *constants* 0 *and* 1, *and predicates* $x > k$ *and* $x \le k$, $k \in \mathbb{N}$, *with the standard interpretation over natural numbers, that we denote* $\mathbb{I}_{\mathtt{cnt}}$.

Regexes with counting may be translated to CA by several methods ([36,33,14,23]). We use a slightly adapted version of [14]—an extension of Glushkov's algorithm [16] to counting. For a regex R, it produces a CA $\mathtt{CA}(R) = (X,Q,\Delta,\{\alpha_0\},F)$. Figure 1 shows an example of such CA. The construction is discussed in detail in [21], here we only overview the important properties needed in Sections 4-6:

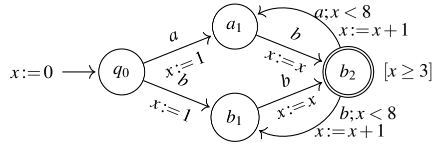

Fig. 1: $\mathtt{CA}(R)$ for $R = \mathtt{((a|b)b)\{3,8\}}$. The accepting condition of all states is $\bot$ except for b_2 whose accepting condition is written in the square brackets.

1. Every occurrence S of a counted sub-expression $T\{\min_S, \max_S\}$ of R corresponds to a unique counter x_S and a substructure A_S of $\mathtt{CA}(R)$. Outside A_S, x_S is inactive (a dead variable) and its value is 0, it is assigned 1 on entering A_S, and every iteration through A_S increments the value of x_S while reading a word from $L(T)$. Our minor modification of [14] is related to the fact that the original assigns 1 to inactive counters while we need 0.
2. $\mathtt{CA}(R)$ has at most $\sharp R+1$ states, $cnt_R.\sharp R^2$ transitions, cnt_R counters. It has at most $\sharp R^2$ transitions if R is flat.
3. $\mathtt{CA}(R)$ has a single initial configuration $\alpha_0 = (q_0,\mathfrak{s}_0)$ s.t. $\mathfrak{s}_0(x_S) = 0$ for each $x_S \in X$.
4. Guards and final conditions are conjunctions consisting of at most one conjunct of the form $\min_S \le x_S$ or $\max_S > x_S$ per counter $x_S \in X$. A transition update may assign to $x_S \in X$ only one of the terms 0, 1, x_S, and x_S+1. It has no guard on x_S if it is assigned x_S, i.e. kept unchanged, it has the guard $x_S \ge \min_S$ iff x_S is reset to 0 or 1 (a counter cannot be reset before reaching its lower bound), and it has the guard $x_S < \max_S$ iff x_S is assigned x_S+1 (counter can never exceed its maximum value $\max_S$). Hence, a counter can never exceed $\max_R$.
5. Flatness of R translates to the fact that configurations of $\mathtt{CA}(R)$ assign a non-zero value to at most one counter. This implies that $\mathtt{Conf}(\mathtt{CA}(R))$ has at most $|Q|.\max_R$ states and also that $\mathtt{CA}(R)$ is *Cartesian*, a property that will be defined in Section 4 and is crucial for correctness of our CA determinization (Theorem 3 in Section 6.)

A DFA can be obtained by the subset construction in the form $\mathtt{DA}(\mathtt{Conf}(\mathtt{CA}(R)))$, called *explicit determinization*. Due to the factor $\max_R$ in the size of $\mathtt{Conf}(\mathtt{CA}(R))$, the explicit determinization is exponential to $\max_R$ even if R is flat, meaning doubly exponential to $|R|$ (R has $\max_R$ written as a decadic numeral). If R is not flat, then the factor $\max_R$ is replaced by $(\max_R)^{cnt_R}$.

4 Counter-subset Construction

In this section, we formulate a modified version of determinization of CA from [36] that constructs a machine of a size independent of $\max_R$. Our version handles the entire class of Cartesian CA (defined below) and in turn also all regexes with flat counting.

The main idea of the determinization remains the same as in [36]. The standard subset construction is augmented with registers, we call them *counting sets*, that can store sets of counter values that would be generated by non-deterministic runs of the CA. The automata with counting-sets as registers are called *counting-set automata*. Our first modification of [36] is indexing of counters by states. In intuitively, this allows to handle cases such as `a*(ba|ab){5}`, where, after reading the first *ab*, the counter is either incremented or not (*b* is the first letter of the counted sub-expression or not). This would violate the uniformity property of CA necessary in [36]—the set of values generated by the non-deterministic CA runs must be the same for every CA state. In our modified version, values at distinct states are stored separately in registers indexed by those states and may differ. Then, in order to handle the indexed counters, we have to introduce a general assignment of counters, allowing to assign the *union* of other counters.[2] Intuitively, when a run non-deterministically branches into several states, each branch needs to continue with its *own copy* of the set, stored in a counter indexed by the state. The union of sets is used when the branches join again. This brings a technical challenge that we solve in this work: how to simulate the counting-set automata fast when the set union and copy are used? The solution is presented in Sections 5 and 6.

Definition 3 (Counting-set automata). *A* counting-set automaton (CSA) *is an automaton with registers operated through the* counting-set language $\Gamma_{\mathtt{set}}$ *under the* number-set interpretation $\mathbb{I}^{\{\}}_{\mathtt{cnt}}$ *where the language* $\Gamma_{\mathtt{set}}$ *extends the counting language* $\Gamma_{\mathtt{cnt}}$ *with the constant* $\emptyset$*, binary union* $\cup$*, and set-filter functions* ∇_p *where* p *is a predicate symbol of* $\Gamma_{\mathtt{cnt}}$*. For simplicity, we restrict terms assigned to counters by transition updates to the form* $t = t_1 \cup \cdots \cup t_n$ *where each* t_i *is either (a) a term of* $\Gamma_{\mathtt{cnt}}$ *or* $\emptyset$*, (b) of the form* $\nabla_{p(t')}$ *where* t' *is a term of* $\Gamma_{\mathtt{cnt}}$*. Each* t_i *is called an* r-term *of* t*.*

The domain of $\mathbb{I}_{\mathtt{set}}$ *is* sets of natural numbers*,* $\mathcal{P}(\mathbb{N})$*. The interpretation of the predicates and functions of* $\Gamma_{\mathtt{cnt}}$ *under* $\mathbb{I}_{\mathtt{set}}$ *is derived from the base number interpretation of the same predicates and functions: A function returns the image of the set in the argument under the base semantics,* $f^{\mathbb{I}_{\mathtt{set}}}(S) = \{f^{\mathbb{I}_{\mathtt{cnt}}}(n) \mid n \in S\}$*. A set satisfies a predicate if some of its elements satisfy the base semantics of that predicate,* $p^{\mathbb{I}_{\mathtt{set}}}(S) \iff \exists e \in S : p^{\mathbb{I}_{\mathtt{cnt}}}(e)$*. Filters then filter out values that do not satisfy the base semantics of their predicate,* $\nabla_p^{\mathbb{I}_{\mathtt{set}}}(S) = \{e \in S \mid p^{\mathbb{I}_{\mathtt{cnt}}}(e)\}$*. Finally,* $\emptyset$ *is interpreted as*

[2] [36] could assign to a counter x only a constant or function of the current value of x.

the empty set and $\cup$ *as the union of sets. We denote memories of the CSA by* $\mathfrak{s}$ *to distinguish them from memories of CA. We write DCSA to abbreviate deterministic CSA.*

Less formally, registers of CSA hold sets of numbers and are manipulated by the increment $x+1$ of all values, assignment of constant sets $\{0\}$, $\{1\}$, and $\emptyset$, denoted by 0, 1, and $\emptyset$, filtering out values smaller or larger than a constant, denoted $\nabla_{x\leq k}(x)$ and $\nabla_{x<k}(x)$, and testing on a presence of a value x satisfying $x \leq k$ or $x < k$, $k \in \mathbb{N}$.

We will present an algorithm that determinizes a CA $A = (X, Q, \Delta, I, F)$, fixed for the rest of the section, into a DCSA $\mathtt{DCSA}(A) = (X^{\{\}}, Q^{\{\}}, \Delta^{\{\}}, I^{\{\}}, F^{\{\}})$. We assume that guards of transitions in Δ and final conditions are of the form $\bigwedge_{x\in Y} p_x[x], Y \subseteq X$, i.e. conjunctions with a at most a single atomic predicate per counter. This is satisfied by all $\mathtt{CA}(R)$, for any regex R (see the list of properties of $\mathtt{CA}(R)$ in Section 3).[3]

Runs of $\mathtt{DCSA}(A)$ will *encode* runs of $\mathtt{DA}(\mathtt{Conf}(A))$ obtained from the explicit determinization of A. Recall that the states $\mathtt{DA}(\mathtt{Conf}(A))$ are sets of configurations of A, pairs $(q, \mathfrak{m})$ of a state and a counter assignment. $\mathtt{DCSA}(A)$ will represent the sets of counter values within a DA state as run-time values of its registers.

Particularly, for every state q and a counter x of the CA, $\mathtt{DCSA}(A)$ has a register x_q in which it remembers, after reading a word w, the set of all values that x reaches in runs of the base CA on w ending in q. Hence, we have $X^{\{\}} = \{x_q \mid x \in X \wedge q \in Q\}$

Definition 4 (Encoding of sets of CA configurations). *A state* $S = \{(q_i, \mathfrak{m}_i)\}_{i=1}^n$ *of* $\mathtt{DA}(\mathtt{Conf}(A))$ *is encoded as the* $\mathtt{DCSA}(A)$ *configuration* $enc(S) = (\{q_i\}_{i=1}^n, \mathfrak{s})$ *where* $\mathfrak{s}(x_q) = \{\mathfrak{m}_i(x) \mid q_i = q\}_{i=1}^n$.

Since a set of assignments appearing with the state q is broken down to sets of values of the individual counters, it disregards relations between values of different counters. For instance, in the DA state $S_1 = \{(q, \{x \mapsto 0, y \mapsto 0\}), (q, \{x \mapsto 1, y \mapsto 1\})\}$, the values of x and y are either both 0 or both 1, but $enc(S_1) = (q, \{x_q \mapsto \{0,1\}, y_q \mapsto \{0,1\}\})$ does not retain this information. It is identical to the encoding of another DA state $S_2 = \{(q, \{x \mapsto 1, y \mapsto 0\}), (q, \{x \mapsto 0, y \mapsto 1\})\}$. This is the same loss of information as in the so-called Cartesian abstraction. The encoding is hence precise and unambiguous only when we assume that inside the states of $\mathtt{DA}(A)$, the relations between counters are always unrestricted—there is no information to be lost. We then call the CA *Cartesian*, as defined below. The encoding function is then unambiguous, and we call the inverse function *decoding*, denoted *dec*.

Definition 5 (Cartesian CA). *Assuming the set of counters of A is* $X = \{x_i\}_{i=1}^m$, *then a set C of configurations of A is* Cartesian *iff, for every state q of A, there exist sets* $N_1, \ldots, N_m \subseteq \mathbb{N}$ *such that* $(q, \{x_i \mapsto n_i\}_{i=1}^m) \in C$ *iff* $(n_1, \ldots, n_m) \in N_1 \times \cdots \times N_m$. *The CA A is* Cartesian *iff all states of* $\mathtt{DA}(\mathtt{Conf}(A))$ *are Cartesian.*

For instance, the DA states S_1 and S_2 above are not Cartesian, while $S_1 \cup S_2$ is.

Similarly as the regex to CA construction of [36], our regex to CA construction discussed in Section 3 returns a Cartesian CA when called on a flat regex.

[3] Every CA can be transformed to this form by transforming the formulae to DNF and creating clones of transitions/states for individual clauses.

Subset construction for Cartesian CA. The algorithm below is a generalization of the subset construction. Let us denote by $\mathsf{index}_q(t)$ the term that arises from t by replacing every variable $x \in X$ by x_q, analogously $\mathsf{index}_q(\varphi)$ for formulas. We have $Q^{\{\}} \subseteq \mathcal{P}(Q)$, the initial configuration $I^{\{\}} = \{enc(I)\}$, and the final conditions assign to $R \in Q^{\{\}}$ the disjunction of the final conditions of its elements, $F^{\{\}}(R) = \bigvee_{q \in R} \mathsf{index}_q(F(q))$.

We will construct $\mathtt{DCSA}(A)$ which is deterministic and its runs encode the runs of DA $\mathtt{DA}(\mathtt{Conf}(A))$. $\mathtt{Conf}(\mathtt{DCSA}(A))$ will be isomorphic to $\mathtt{DA}(\mathtt{Conf}(A))$. For that, we need for each transition δ of $\mathtt{DA}(\mathtt{Conf}(A))$ one unique transition of $\mathtt{DCSA}(A)$ over the same letter enabled in the encoding of the source of δ and generating the encoding of the target of δ. In other words, we need for each transition $dec(R,\mathfrak{s}) \xrightarrow{\{a\}} dec(R',\mathfrak{s}')$ of $\mathtt{DA}(\mathtt{Conf}(A))$ one unique transition $\delta' = R \xrightarrow{\{a,\varphi,u\}} R' \in \Delta^{\{\}}$ with $(R',\mathfrak{s}') = \delta'(R,\mathfrak{s})$. That transition δ' will be built by summarizing the effect of all base CA a-transitions enabled in the CA configurations of $dec(R,\mathfrak{s})$.

To construct the transition δ', we first translate each base transition $\delta = q \xrightarrow{\{a,\varphi_\delta,u_\delta\}} r \in \Delta$ into its set-version $\delta^{\{\}}$, supposed to transform an encoding of a (Cartesian) set C of configurations, $enc(C)$, into the encoding of the set of their images under δ, $enc(\delta(C))$, and enabled if δ is enabled for at least one configuration in C. To that end, assuming $\varphi_\delta = \bigwedge_{x \in X} p_x[x]$, we (1) construct the update u^{∇}_δ from u_δ by substituting in every $u_\delta(x), x \in X$ variables $y \in X$ by their filtered versions $\nabla_{p_y}(y)$, (2) add indices to registers that mark the current state, resulting in the transition $\delta^{\{\}} = q \xrightarrow{\{a,\varphi^{\{\}}_\delta,u^{\{\}}_\delta\}} r$ where $\varphi^{\{\}}_\delta = \mathsf{index}_q(\varphi_\delta)$ and $u^{\{\}}_\delta$ assigns to every $x_r, x \in X$ the term $\mathsf{index}_q(u^{\nabla}_\delta(x))$.

The states $Q^{\{\}}$ and the transitions $\Delta^{\{\}}$ are then constructed as the smallest sets satisfying that $enc(I) \in Q^{\{\}}$ and every $R \in Q^{\{\}}$ has for every $a \in \Sigma$ the outgoing transitions constructed as follows. Let $\{q_j \xrightarrow{\{a,\varphi_j,u_j\}} r_j\}_{j \in J}$ for some index set J be the set of *constituent a-transitions* for R, all a-transitions $\delta^{\{\}}$ where $\delta \in \Delta$ originates in R. To achieve determinism, $\Delta^{\{\}}$ has the transition $R \xrightarrow{\{a,\psi,u\}} R'$ for every minterm $\psi \in Minterms(\{\varphi_j\}_{j \in J})$. The update u and target R' are constructed from the set $\{q_j \xrightarrow{\{a,\varphi_j,u_j\}} r_j\}_{j \in K}$, $K \subseteq J$, of constituent transitions with guards φ_j compatible with the minterm ψ, i.e., with satisfiable $\psi \wedge \varphi_j$. R' is the set of their target states, $R' = \{r_j\}_{j \in K}$, and $u(x)$ unites all their update terms $u_j(x)$, i.e. $u(x) = \bigcup_{j \in K} u_j(x)$, for each $x \in X^{\{\}}$.

Example 1. When showing examples of transition updates, we write $x := t$ to denote that $u(x) = t$ and we omit the assignments $x := \emptyset$ in CSA.

Let $R = \{p,q\}$ and let the a-transitions originating at R be $q \xrightarrow{\{a,\top,x:=x\}} s$, $p \xrightarrow{\{a,x<n,x:=x+1\}} r$, and $p \xrightarrow{\{a,x \geq m,x:=1\}} s$. They induce three constituent transitions for R and a, $q \xrightarrow{\{a,\top,x_s:=x_q\}} s$, $p \xrightarrow{\{a,x_p<n,x_r:=\nabla_{x<n}(x_p)+1\}} r$, and $p \xrightarrow{\{a,x_p \geq m,x_s:=1\}} s$. A transition $R \xrightarrow{\{a,\psi,u'\}} R'$ is constructed for each of the following minterms ψ: $x_p{<}n \wedge x_p{\geq}m$, $\neg x_p{<}n \wedge x_p{\geq}m$, $x_p{<}n \wedge \neg x_p{\geq}m$, $\neg x_p{<}n \wedge \neg x_p{\geq}m$. For the first one, all three constituent transitions are compatible and so the update u' is $x_r := \nabla_{x<n}(x_p) + 1; x_s := x_q \cup 1$ (update of x_r is taken from the first constituent transitions leading to r, update of x_s is the union of the updates of the second two transitions leading to s) and the target state is $R' = \{r,s\}$. □

$\mathtt{DCSA}(A)$ is deterministic since it has a single initial configuration and the guards of transitions originating in the same state are minterms. The size of $\mathtt{DCSA}(A)$ obviously depends only on the size of A and not on the interpretation of the language. Especially,

when A is $\mathtt{CA}(R)$ for some regex R, the size does not depend on $\max_R$. The theorem below is proved in [21].[4]

Theorem 1. $\mathtt{DCSA}(A)$ *is deterministic,* $|\mathtt{DCSA}(A)| \in O(2^{|A|})$*, and if A is Cartesian, then* $L(A) = L(\mathtt{DCSA}(A))$.

Since for regexes with flat counting, our regex to CA algorithm always returns a Cartesian CA, we can transform them into DCSA.

5 Fast Simulation of Counting-set Automata

In this section, we discuss how a run of a DCSA on a given word can be *simulated* efficiently to achieve fast matching. Let us fix a word $w = a_1 \cdots a_n$ together with the DCSA $A = (X, Q, \Delta, \{\alpha_0\}, F)$. We wish to construct the run of the DCSA on w and test whether the reached configuration is accepting. We aim at a running time linear to $|w|$ and independent of the sizes of the sets stored in A's registers at run-time.

We will assume that the initial configuration α_0 of A assigns to every register a singleton or the empty set. The assumption is satisfied by CSA constructed from $\mathtt{CA}(R)$, R being any regex, by the algorithms of Section 4 and also Section 6.[5]

Technically, the simulation maintains a configuration $\alpha = (q, \mathfrak{s})$, initialized with α_0, and for every i from 1 to n, it constructs the transition $\alpha \text{-}\{a_i\}\text{→} \alpha'$ of $\mathtt{Conf}(A)$ and replaces α by the successor configuration $\alpha' = (q', \mathfrak{s}')$. We use the key ingredient of fast simulation from [36], the *offset-list data structure* for sets of numbers with constant time addition of 0/1, comparison of the maximum to a constant, reset, and increment of all values. The problem is that the newly added union and copy of sets are still linear to the size of the sets, and hence linear to the maximum counter bounds. We show how, under a condition introduced below, set copy can be avoided entirely and the cost of union can be amortized by the cost of incrementing the sets. This will again allow a CSA-simulation in time independent of $\max_A$ and falling into $O(|A| \cdot |w|)$.

First, we define a property of CSA sufficient for fast simulation—that the updates on its transitions do not *replicate counters*.

Definition 6 (Counter replication). *We say that a CSA* replicates counters *if for some transition $q \text{-}\{a,\varphi,u\}\text{→} r$, some counter appears in the image of u twice, that is, it appears in two r-terms of some $u(x)$ or it appears in $u(x)$ as well as in $u(y)$ for $x \neq y$. A* non-replicating *CSA does not replicate counters.*

For instance, $\{x \mapsto x; y \mapsto x+1\}$ and $\{x \mapsto x \cup x+1, y \mapsto y\}$ are updates where x is replicated, $\{x \mapsto x+1, y \mapsto y\}$ is not a replicating update.

[4] It may be interesting to note that, as follows from our formulation of the determinization, the construction is independent of the particular f.o.l. used to manipulate registers and of its interpretation. The determinization could be applied to any kind of automata that fits the definition of automata with registers. The numbers could be manipulated by other functions and tests, natural numbers could be replaced by reals etc. The counting-set automata are themselves an instance of automata with registers. One could also think about push-down automata or, with small modifications, variants of data-word automata with registers.

[5] This is a technical assumption important in order for unions of the initial sets not to influence the overall complexity of the simulation.

Offset-list data structure. The *offset-list* data structure of [36] allows constant time implementation of the set operations of increment of all elements, reset to $\emptyset$ or $\{0\}$ or $\{1\}$, addition of 0 or 1, and comparison of the maximum with a constant.

It assigns to every counter $x \in X$ a pointer $ol(x)$ to an *offset-list pair* (o_x, l_x) with the *offset* $o_x \in \mathbb{N}$ and a sorted list $l_x = m_1, \ldots, m_k$ of integers. The data structure implementing the list needs constant access to the first and the last element, forward and backward iteration of a pointer, and insertion/deletion at/before a pointer to an element. This is satisfied for instance by a doubly-linked list that maintains pointers to the first and the last element. The offset-list pair represents the set $\mathfrak{s}(x) = \{m_1 + o_x, \ldots, m_k + o_x\}$. Union of two such sets is still linear in their size, but we will show that if the CSA does not replicate counters, the cost of set unions can be amortized by the cost of increments.

Finding the CSA transition and evaluating the update. The first step of computing α' from α is finding the transition $q \text{-}\{a_i,\varphi,u\}\!\!\rightarrow q' \in \Delta$, the only a_i-transition from q that is enabled, i.e. where $\mathfrak{s} \models \varphi$. The simplest algorithm iterates through the transitions of Δ and, for each of them, tests whether $\mathfrak{s}$ satisfies its guard. The cost of evaluating an atomic counter predicate p, i.e., deciding whether $\mathfrak{s} \models p$, is constant: since the lists l_x are sorted, we only need to access the first or the last element and the offset to decide $x < n$ or $x \geq n$, respectively. With that, the cost of evaluating φ is linear to the size of φ. The cost of the iteration through the transitions of Δ is then linear in the sum of their sizes, which is within $O(|A|)$.

Having found $q \text{-}\{a_i,\varphi,u\}\!\!\rightarrow q'$, we evaluate its update to compute $\mathfrak{s}'$ and compute α' as $(q', \mathfrak{s}')$. We will explain the algorithm and argue that the amortized cost of computing $\mathfrak{s}'$ is in $O(|X|)$. The update is evaluated by, for each $x \in X$, evaluating all r-terms in $u(x)$, uniting the results, and assigning the union to $ol(x)$.

First, we argue that evaluating an r-term t of $u(x)$, i.e. computing $t(\mathfrak{s})$, is amortized constant time. Since the counters are non-replicating, we can compute the value of each r-term $t[y]$ in situ. That is, we modify the offset-list pair (o_y, l_y) and return the pointer $ol(y)$. The original value of y can be discarded after evaluating $t[y]$ since y does not appear in any other r-term. There are 5 cases: (1) If t is 0 or 1, then we return a pointer to a fresh offset-list pair with the offset 0 and the list containing only 0 or 1, respectively. This is done in constant time.

(2) If t is $y \in Y$, then we return $ol(y)$.

(3) If t is $y + 1$, then o_y is incremented by one. This constant time implementation of the increment is the reason for pairing the lists with the offsets.

(4) If t is $\nabla_p[y]$, then l_y is filtered by the atomic predicate p. Filtering with the predicate $x \geq n$ uses the invariant of sortedness of l_y. It is done by iterating the following steps: i) test whether the list head is smaller than $n - o_y$ and ii) if yes, remove the head, if not, terminate the iteration. Every iteration is constant time: The cost of the iterations which remove an element is amortized by the cost of additions of the element to the list. What remains is only the constant cost of the last iteration which detects an element greater or equal to $n - o_y$, or that the list is empty. Filtering with $x < n$ is analogous (the iterations test and remove the last element instead of the head).

(5) If t is $\nabla_p(y) + 1$, then the construction for the constant increment is applied after the constant filter discussed above.

Next, we argue that computing the union of values of the r-terms in $u(x)$ may be amortized by the cost of evaluating the increment terms. Let $l_1,\dots,l_n$ be the offset-list representations of the values of the terms in $u(x)$ computed by the algorithm above. The offset-list representation of their union is computed by a sequence of merging, as $merge(l_1, merge(l_2, \dots merge(l_{n-1}, l_n)\dots))$. Particularly, given two pointers to offset-lists l, l', $merge(l, l')$ implements their union: it chooses the offset-list that represents a set with the larger maximum, assume that it is l, and inserts the elements represented by the other list, l', to it. We say that l' *is merged into* l. This is done by the standard sorted-list merging in time $O(|l'|)$ where $|l'|$ is the length of l'. Since l' is without duplicities and with minimum 0, $O(|l'|) \subseteq O(\max(l'))$ where $\max(l')$ is the maximal element.

The $O(\max(l'))$ cost is amortized by the cost of evaluating increments. The offset-list pair at l' has seen at least $\max(l') - 1$ increments since the only elements inserted into it are 0, 1, or, during merge, elements from other sets smaller than $\max(l')$. These increments of l' are the budget used to pay for the mergeing of l' into l. After the merge, the offset-list pair of l' is discarded (as the CSA is non-replicating, it is no longer needed) hence the budget is used only once. Last, the assignment of the union to c is done by a constant time assignment of a pointer to the offset-list returned by the merge.

Overall complexity of the simulation. Let us define the cost $cost(x)$ of manipulations with the counter $x \in X$ during one step of the simulation as the sum of the costs of: (1) evaluating all r-terms containing c, (2) merging their offset-list into other ones, (3) creating offset-lists for terms 0 or 1 in $u(x)$ and merging them into other offset-lists, (4) the assignment of the result of $u(x)$ to x. The cost of processing a single letter a_i is then the sum $\sum_{x\in X} cost(x)$ and $|w| \cdot \sum_{x\in X} cost(x)$ is the cost of the entire simulation. Since the CSA is non-replicating and evaluating a single r-term is amortized constant time, the cost of (1) is in amortized constant time. The cost of (2) is amortized by increments from step (1). The creation and insertion of singletons in (3), at most two in $u(x)$, is constant time. The pointer assignment in (4) is constant time. The $cost(x)$ is therefore amortized constant time, the amortized time of evaluating the update u is in $O(|X|)$, and the cost of the updates through the simulation is in $O(|X| \cdot |w|)$. The cost of choosing the transitions, by evaluating their guards, is in $O(|A| \cdot |w|)$ by the above analysis. Analogously, the cost of testing the accepting condition at the reached configuration is in $O(|A|)$.

Theorem 2. *If A is non-replicating, then its simulation on w takes $O(|A| \cdot |w|)$ time.*

6 Augmented Determinization

In this section, we augment the subset construction from Section 4 with optimizations that prevent counter replication and hence extend the class of regexes that can be matched fast by simulation of the CSA. It optimizations are tailored to CA with the special properties of $\mathtt{CA}(R)$, for a regex R, listed in Section 3.

Intuition for the optimizations. The emergence of counter replication and means of its elimination in the augmented construction, by techniques of *counter sharing* and *increment postponing*, are illustrated on simplified fragments of CA in Figure 2.

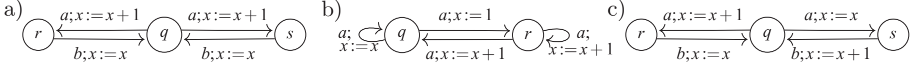

Fig. 2: Sub-structures of CA that are sources of counter replication.

In a), $\mathtt{DCSA}(\mathtt{CA}(R))$ has transitions $\{q\} \xrightarrow{a, x_r := x_q + 1, x_s := x_q + 1} \{r, s\} \xrightarrow{b, x_q := x_r \cup x_s} \{q\}$. The first transition replicates the entire content of the x_q, the second one unites the two sets. Both transitions are expensive. The can be optimized by detecting that the values of x_s and x_r are the same, being generated by *syntactically identical* updates, and storing the values in a *shared counter* $x_{\{s,r\}}$. This would result in transitions $\{q\} \xrightarrow{a, x_{\{r,s\}} := x_{\{q\}} + 1} \{s, t\} \xrightarrow{b, x_{\{q\}} := x_{\{r,s\}}} \{q\}$, with the replication and union eliminated.

Figure b) then illustrates why a counter x_P, $P \subseteq Q$, represents the set of values shared between the original counters x_p, $p \in P$. That is, x_P does not always hold the entire sets stored in the counters $x_p, p \in P$. If their values are not the same, it stores only their intersection. The value of each x_p is then partitioned among several shared counters x_S with $p \in S$. In b), $\mathtt{DCSA}(\mathtt{CA}(R))$ has transitions $q \xrightarrow{a, x_q := x_q ; x_r := 1} \{q, r\} \xrightarrow{a, x_q := x_q \cup x_r + 1 ; x_r := 1 \cup x_r + 1} \{q, r\}$, replicating the counter x_r. Counter sharing would then generate transitions $q \xrightarrow{a, x_{\{q\}} := x_{\{q\}} ; x_{\{r\}} := 1} \{q, r\} \xrightarrow{a, x_{\{q\}} := x_{\{q\}} ; x_{\{r\}} := 1 ; x_{\{q,r\}} := x_{\{r\}} + 1} \{q, r\}$ with counters $x_{\{q\}}$, $x_{\{r\}}$ for the subsets exclusive to x_q and x_r, respectively, and $x_{\{q,r\}}$ for the intersection.

Last, in c), we illustrate the technique of *increment postponing*. $\mathtt{DCSA}(\mathtt{CA}(R))$ would have transitions $\{q\} \xrightarrow{a, x_r := x_q + 1, x_s := x_q} \{s, t\} \xrightarrow{b, x_q := x_r \cup x_s + 1} \{q\}$. Since the increments on the two branches happen in different moments, the values of x_r and x_s differ until the last increment of x_s synchronizes them. We avoid replication by storing the non-incremented value, obtained from x_q, in a counter shared by x_r and x_s and remembering that an increment of x_r has been postponed. This is marked with + in the name of the shared counter $x_{\{r^+,s\}}$. When the values of x_r and x_s synchronize (the increment is applied to x_s too), the postponed increment is evaluated and the +-mark is removed. We would create transitions $\{q\} \xrightarrow{a, x_{\{r^+,s\}} := x_{\{q\}}} \{s, t\} \xrightarrow{b, x_{\{q\}} := x_{\{r^+,s\}} + 1} \{q\}$. If, before the synchronization, the value of the marked counter is either tested or incremented for the second time, we declare an *irresolvable replication* and abort the entire construction (we allow postponing of only one increment). To prevent this situation from arising needlessly, we let states remember the counters that must have the empty value and we ignore these counters.

Augmented Determinization Algorithm. The augmented determinization produces from $\mathtt{CA}(R) = (X, Q, \Delta, \{\alpha_0\}, F)$ the CSA $\mathtt{DCSA}^{\mathsf{a}}(\mathtt{CA}(R)) = (X^{\mathsf{a}}, Q^{\mathsf{a}}, \Delta^{\mathsf{a}}, \{\alpha_0^{\mathsf{a}}\}, F^{\mathsf{a}})$. Its counters in X^{a} are of the form x_S where $x \in X$ and $S \subseteq Q^+$ and $Q^+ = Q \cup \{q^+ \mid q \in Q\}$. The guiding principle of the algorithm is that an assignment $\mathfrak{s}^{\mathsf{a}}$ of X^{a} represents an assignment $\mathfrak{s}$ of the counters in $X^{\{\}}$ of $\mathtt{DCSA}(\mathtt{CA}(R))$, namely, for each $x_q \in X^{\{\}}$,

$$\mathfrak{s}(x_q) = \bigcup_{q \in S, S \subseteq Q^+} \mathfrak{s}^{\mathsf{a}}(x_S) \cup \bigcup_{q^+ \in S, S \subseteq Q^+} \{ n + 1 \mid n \in \mathfrak{s}^{\mathsf{a}}(x_S) \}. \tag{1}$$

We will use some simplifying notation. As discussed in Section 3, by the construction of $\mathtt{CA}(R)$, the increment of c and the guard $x < \max_x$ always appear on its transitions

together, without any other guard on x. Hence, in $\mathtt{DCSA}(\mathtt{CA}(R))$, all terms with an increment or filtering are of the form $\nabla_{x<\max_x}(x_{q^\circ})+1$. We will denote them by the shorthand $x_{q^\circ} \oplus 1$ (we are using q° to denote an element from the set Q^+, either q or q^+, for $q \in Q$).

The states of $\mathtt{DCSA}^{\mathsf{a}}(\mathtt{CA}(R))$ will additionally be distinguished according to which of the counters of X^{a} are *active*, i.e., could have a non-empty value. Counters always valued by $\emptyset$ can be ignored, which simplifies transitions and decreases the chance of an irresolvable counter replication. The states of $\mathtt{DCSA}^{\mathsf{a}}(\mathtt{CA}(R))$ are thus of the form (R, Act) where $R \subseteq Q$ and $Act \subseteq X^{\mathsf{a}}$ is a set of active counters.

The initial configuration is $\alpha_0^{\mathsf{a}} = ((\{q_0\}, \{x_{\{q_0\}} \mid x \in X\}), \mathfrak{s}_0^{\mathsf{a}})$ where $\mathfrak{s}_0^{\mathsf{a}}$ assigns $\{0\}$ to every $x_{\{q_0\}}, x \in X$ and $\emptyset$ to every other counter in X^{a}. The final condition assignment $F^{\mathsf{a}}((R, Act))$ is, for each $(R, Act) \in Q^{\mathsf{a}}$, constructed from $F^{\{\}}(R)$ by replacing every predicate $p[x_q]$ by the disjunction $p[x_q]^{Act} = \bigvee_{x_S \in Act, q \in S} p[x_S]$ that encodes $p[x_q]$ using the counters of Act in the sense of (1).

The transitions in Δ^{a} are constructed from transitions in $\Delta^{\{\}}$. For source state $(R, Act) \in Q^{\mathsf{a}}$, an original transition $R \xrightarrow{a,\varphi,u} R' \in \Delta^{\{\}}$, and set of active counters $Act \subseteq X^{\mathsf{a}}$, Δ^{a} has the transition $(R, Act) \xrightarrow{a,\varphi^{\mathsf{a}},u^{\mathsf{a}}} (R', Act')$, constructed as follows:

The guard φ^{a} is made from φ by replacing every predicate $p[x_q]$ by the equivalent version with shared counters $p[x_q]^{Act}$ (as when constructing F^{a} above).

The update u^{a} is constructed in three steps. First, the update $u^{\mathtt{sh}}$ is made from u by expressing the r-terms of u using the shared counters X^{a}. Each $t[x_q]$ is replaced by

$$t^{\mathsf{a}} = \bigcup \Big(\{\, t[x_S] \mid x_S \in Act, q \in S \,\} \cup \{\, t[x_S] \oplus 1 \mid x_S \in Act, q^+ \in S \,\} \Big) .$$

Notice that all postponed increments are *evaluated* in $u^{\mathtt{sh}}$, transformed to normal increments. If $u^{\mathtt{sh}}$ has an r-term $t \oplus 1 \oplus 1$, i.e., a double increment, then the whole construction aborts and declares an *irresolvable counter replication*. We allow postponing only one increment.[6] Otherwise, we proceed to resolve counter replication. First, we make sure that every counter appears in the image of the update only in one kind of r-term. We collect the set *Conflict* of all r-terms $x_S \oplus 1$ of $u^{\mathtt{sh}}$ with *conflicting increments*, i.e. such that also x_S is an r-term of $u^{\mathtt{sh}}$. In update u^+, conflicting increments are *postponed*. For $x \in X$, $q \in Q$, and $u^{\mathtt{sh}}(x_q) = \bigcup T$,

$$u^+(x_q) = \bigcup \big(T \setminus \mathit{Conflict}\big) \;\text{ and }\; u^+(x_{q^+}) = \bigcup \{\, x_S \mid x_S \oplus 1 \in T \cap \mathit{Conflict} \,\} .$$

The final update u^{a} then resolves counter replication, by grouping r-terms replicated in u^+ under a common l-value (we call z an *l-value* of r-terms of $u^+(z)$). For an r-term t of u^+, let $\mathtt{lval}(t)$ be the set of its l-values. Note that $\mathtt{lval}(t)$ is always of the form $\{x_{q^\circ}\}_{x \in S}$ for some fixed $x \in X$ (see property 4 of $\mathtt{CA}(R)$ in Section 3). We let Act' be the set of counters x_S with $\mathtt{lval}(t) = \{x_{q^\circ}\}_{x \in S}$ for some r-term of u^+. For all $x_S \in X^{\mathsf{a}}$, if $x_S \notin Act'$ then $u^{\mathsf{a}}(x_S) = \emptyset$ else

$$u^{\mathsf{a}}(x_S) = \bigcup \{\, t \mid t \text{ is an r-term of } u^+ \text{ and } \mathtt{lval}(t) = \{x_{q^\circ}\}_{q^\circ \in S} \,\} .$$

[6] Also transition guards and final conditions of $\mathtt{DCSA}^{\mathsf{a}}(\mathtt{CA}(R))$ must not contain the +-mark since evaluating them regardless the postponed increments would return incorrect results. However, declaring counter replication on seeing a double increment here covers these cases due to the structural properties of $\mathtt{CA}(R)$.

Example 2. Let us have $R \xrightarrow{a,\varphi,u} R' \in \Delta^{\emptyset}$ created in Example 1 with $R = \{p,q\}$, $R' = \{r,s\}$, $\varphi = x_p < n \wedge x_p \geq m$, and $u = \{x_r := x_p \oplus 1, x_s := x_q \cup 1\}$. Let $Act = \{x_{\{p,q\}}, x_{\{p,q^+\}}\}$. Then $u^{\mathtt{sh}} = \{x_r := x_{\{p,q^+\}} \oplus 1 \cup x_{\{p,q\}} \oplus 1, x_s := x_{\{p,q^+\}} \oplus 1 \cup x_{\{p,q\}} \cup 1\}$. Note that the x_q in $u(x_s)$ becomes $x_{\{p,q^+\}} \oplus 1$, corresponding to the right part of the definition of $t^{\mathtt{a}}$ (the postponed increment x_{q^+} is evaluated in $u^{\mathtt{sh}}$). Note that the r-term $x_{\{p,q\}} \oplus 1$ is in *Conflict* as $x_{\{p,q\}}$ is an r-term of $u^{\mathtt{sh}}$ too. Therefore it is postponed in u^+, i.e. $u^{\mathtt{sh}}(x_r) = x_{\{p,q\}} \oplus 1 \cup \cdots$ becomes $u^+(x_{r^+}) = x_{\{p,q\}}$. We get $u^+ = \{x_r := x_{\{p,q^+\}} \oplus 1, x_s := x_{\{p,q^+\}} \oplus 1 \cup x_{\{p,q\}} \cup 1, x_{r^+} := x_{\{p,q\}}\}$. Finally, $u^{\mathtt{a}}$ groups r-terms replicated in u^+ under a common l-value: $u^{\mathtt{a}} = \{x_{\{r,s\}} := x_{\{p,q^+\}} \oplus 1, x_{\{s\}} := 1, x_{\{s,r^+\}} := x_{\{p,q\}}\}$. The next active counters are $Act' = \{x_{\{r,s\}}, x_{\{s\}}, x_{\{s,r^+\}}\}$. Note that, for $x_{\{p,q^+\}}$, the postponed increment at p^+ was synchronized on this transition, while the conflict at $x_{\{p,q\}}$ was solved by postponing increment and marking r with $^+$. □

The algorithm either returns the CSA $\mathtt{DCSA^a}(\mathtt{CA}(A))$, or detects an irresolvable counter replication, in which case $\mathtt{DCSA^a}(\mathtt{CA}(A))$ does not exist.[7] Let $m = \sharp R$ and recall that n denotes the length of the matched text, $|w|$. Since $\mathtt{CA}(R)$ has at most m states and m^2 transitions, a basic analysis of the algorithm's data structures reveals that the resulting CSA has at most 2^{2^m} states, each with at most 2^{m^2} outgoing transitions, each transition of the size in $O(m2^m)$. Because $\mathtt{DCSA^a}(\mathtt{CA}(A))$ encodes $\mathtt{DCSA}(\mathtt{CA}(A))$, it has the same language, and it also inherits its determinism. Since it does not replicate counters, it can be simulated in pattern matching fast, in time linear to the text and independent of the counter bounds. The following theorem is proved in [21].

Theorem 3. *For R with flat counting, if $\mathtt{DCSA^a}(\mathtt{CA}(R))$ exists, then it does not replicate counters, its size is in $O(2^{2^m} m)$, $L(\mathtt{CA}(R)) = L(\mathtt{DCSA^a}(\mathtt{CA}(R)))$, and it can be simulated on a word w of the length n in time $O(2^{2m} mn)$.*

Matching can be done in time of constructing the CSA plus its simulation, which in the sum is indeed fast, not dependent on k and linear in n. It can also be noted that the m in the exponents above is not the size of the entire regex, but only the size of the counted sub-regexes.

7 Regexes with Synchronizing Counting

Finally, in this section we define the class of regexes with synchronizing counting, which precisely captures when the CSA created by our construction in Section 6 does not replicate counters and hence allow fast matching (in the sense of Theorem 3).

Definition 7 (Regexes with synchronizing counting). *A regex has* synchronizing counting *iff it has no sub-expression $S\{\mathtt{n},\mathtt{m}\}$ where for some $k \in \mathbb{N}$, a word from $L(S)^k$ has a prefix from $L(S)^{k+1}$.*

For instance, `(ac*){1,4}(ab|ba){3,5}(a(ab)*){2,8}` is a regex with synchronizing counting as each word from $L(\mathtt{ac*})^k$ must contain the symbol a exactly k times,

[7] Aborting the construction here simplifies the description, but it would also be possible to continue the construction and return a DCSA that does not guarantee fast simulation.

words from $L(\texttt{ab|ba})^k$ must have exactly $2k$ symbols, and words from $L(\texttt{a(ab)*})^k$ can be uniquely split at the first a in the `a(ab)*`. In comparison, `(a|aa){2,5}` does not have synchronizing counting as $a \cdot a \cdot a$ is a prefix of $aa \cdot aa$.

Intuitively, there is no pair of paths through $\texttt{CA}(S\{\texttt{m,n}\})$ starting at the same state, over the same word, ending in the same state, where the number of increments differs by two. In such case, $\texttt{DCSA}^{\texttt{a}}(\texttt{CA}(S\{\texttt{m,n}\}))$ would have to delay two increments, which our construction does not allow. The theorem below is proved in [21].

Theorem 4. *Given a regex R with flat counting, the algorithm of Section 6 returns* $\texttt{DCSA}^{\texttt{a}}(\texttt{CA}(R))$ *if and only if R has synchronizing counting.*

Corollary 1. *Regexes with flat synchronizing counting have a fast matching algorithm.*

Proof. From Theorems 3 and 4.

Counting with Markers. Even though designing and recognizing synchronizing counting is usually intuitive, it may also be tricky. For instance, `(\\\\d+\\\\.){3}`, from the database of real-world regexes we use in our experiment, has synchronizing counting, while `ICE_Dims.{92}((_?(X|\d+)){13})` does not.[8] A vast majority of real-world regexes we examined fortunately belong to very easily recognizable subclasses of synchronizing counting. The most wide-spread and easy to recognize are regexes with *letter-marked counting*, where every sub-expression $S\{\texttt{m,n}\}$ has a set of marker letters such that every word from $L(S)$ has exactly one occurrence of a marker letter. [9]

Marker letters may be generalized to *marker words*, though, markers that can arise by concatenation of several words from $L(S)$ cannot be used. The condition that has to be satisfied is that any word from $L(S)^k$, $k \in \mathbb{N}$, has exactly k non-overlapping occurrences of marker words as infixes. Another sufficient property of S is that it has words of a *uniform length*. The idea of markers may be generalized further until the point when the set of marker words is specified by general regexes, when we get precisely the synchronizing counting. The regexes with letter-marked counting are easily human as well as machine recognizable (see a simple $O(|R|^2)$-time algorithm in [21]).

8 Practical Considerations

Although the main point of this work is the theoretical feasibility of fast matching with synchronizing counting, we will also argue that the results are of practical relevance. To this end, we show experimentally that synchronizing counting and marked counting cover a majority of practical regexes. We also give arguments that matching with the CSA constructed in Section 6 can be done efficiently.

[8] An automated way of identifying synchronizing counting would be running the CSA-to-DCSA determinization from Section 6, but this is exponential to $|R|$.

[9] That letter-marked counting is a strict superset of the class that is in [36] conjectured as handled by the algorithm of [36]. The conjecture of [36] is also not correct, as shown in [21].

8.1 Occurrence of Synchronizing Counting in Practice

To substantiate the practical relevance of synchronizing counting regexes, we examined a large sample of practical regexes using a simple checker of letter-marked counting. The benchmark consists of over 540 000 regexes collected from (1) a large scale analysis of software projects [10]; (2) regexes used by network intrusion detection systems Snort [27], Bro [29], Sagan [34], and the academic papers [42,38]; (4) the RegExLib database of regexes [28].

From the regexes that we could parse[10], 31 975 contained counting. We selected those with flat counting and with the sum of upper bounds of counters larger than 20 (as was done in [36] to filter out counting with small bounds that can be handled through counter unfolding and traditional methods)[11]. This left us with 5 751 regexes. From these, only 46 regexes (0.8 %) have counting that is not letter-marked. Furthermore, we manually checked these regexes and we identified that 22 of them have synchronizing counting. We have therefore found only 24 regexes with non-synchronizing counting, i.e., 0.4 % of the examined set of regexes with flat counting.

The 24 non-synchronizing regexes are listed in [21]. Some of them may clearly be rewritten with synchronizing counting, such as `(.+){25}(.*)`, which can be rewritten as `.{25,}(.*)`. We speculate that some of them might in fact represent a mistake, such as `(.*){1,32000}[bc]` where the counter matches the empty word, or `(\n\s+)(criterion .*\n)(\s.+){1,99}` where the `\s.+` might have been intended as `\s\S+` (`\s` are white spaces, `\S` are all the other characters). Synchronizing counting seems to capture the intuition with which counting is often written, hence reporting non-synchronizing counting might help identifying bugs.

By the same methodology and from a nearly identical benchmark, [36] arrived to a sample of 5 000 regexes with flat counting with the sum of bounds larger than 20. The algorithm of [36] did not cover 571 regexes from the 5 000, which is 11 % of the examined set of regexes with flat counting (in contrast to the 0.4 % with non-synchronizing counting and the 0.8 % with counting that is not letter-marked, measured on a slightly larger set of regexes). The two sets of regexes with flat counting, the 5 751 of ours and the 5 000 of [36], are not perfectly identical, however. Differences are to a small degree caused by differences in the base database ([36] uses about 18 more regexes that are proprietary and excludes 26 regexes with counter bounds larger than 1 000), and to a larger degree by small differences in the parsers.

8.2 Practical Efficiency of Matching with Synchronizing Counting

The size and the worst-case time of simulation of $\mathtt{DCSA}^{\mathsf{a}}(\mathtt{CA}(R))$ are still exponential to the number of states of $\mathtt{CA}(R)$ (namely, $O(2^{2^m}m)$ and $O(2^{2m}mn)$ where $m = \sharp R$ equals the number of states of $\mathtt{CA}(R)$, cf. Theorem 3). The potential problem is that the algorithm may generate at most 2^m counters, and this potentially threatens practicality of our matching algorithm.

[10] We did not parse 38 558 regexes since their syntax was broken or contained some advanced features we do not support.

[11] 926 regexes contain nested counting and 25297 regexes contain small upper bounds.

First, it should be noted that the m in the exponent can be decreased from the size of the entire regex to the size of the counted sub-expression, which is usually very small. Then, although an efficient implementation is beyond the scope of this paper and we are leaving it as a future work, we give some indirect arguments for practicality of the CA-to-CSA algorithm.[12]

By the standard techniques of register allocation [1], it is possible to decrease the number of counters and counter assignments other than identity dramatically. In fact, simply eliminating needless renaming of counters and reusing the same name whenever possible, our algorithm creates CSA isomorphic to those of [36] when run on regexes handled by [36]. The work [36] already shows that simulating these CSA may be done efficiently and that it brings dramatic improvements over best matchers on counting-intensive examples.

In our experience with hand-simulating the algorithm on practical examples, cases not handled by [36] do not behave much differently, and the numbers of CSA counters do not have a strong tendency to explode.

9 Conclusions

We have extended the regex matching algorithm of [36] and shown that the extended version allows fast pattern matching of so-called synchronising regexes, a class of regexes that we have newly introduced. The class of synchronising regexes significantly extends all previously known classes of regexes that allow fast matching and covers a majority of regexes appearing in practice (wrt. our empirical study).

In the future, we plan to study extensions of the presented techniques to regexes with nested counting (non-flat). This will probably require a more sophisticated alternative of the offset-list data structure for sets, capable of storing relations of numbers. An interesting question is also how and when regexes can be rewritten to a synchronizing form and for what cost.

Acknowledgment

This work has been supported by the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme, the Czech Science Foundation project 23-06506S, and the FIT BUT internal project FIT-S-23-8151.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley (August 2006), http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0321486811

[12] A competitive matcher that runs on real-world regexes requires an extensive infrastructure, optimized data structures for the shared registers, and ideally an on-the-fly version of the CA-to-CSA determinization (similar to the online DFA simulation).

2. Antimirov, V.: Partial derivatives of regular expressions and finite automaton constructions. Theoretical Computer Science **155**(2), 291 – 319 (1996). https://doi.org/10.1016/0304-3975(95)00182-4, https://doi.org/10.1016/0304-3975(95)00182-4
3. Baldwin, A.: Regular expression denial of service affecting express.js. https://medium.com/node-security/regular- expression-denial-of-service-affecting-express-js-9c397c164c43 (2016)
4. Björklund, H., Martens, W., Timm, T.: Efficient incremental evaluation of succinct regular expressions. In: CIKM'15. ACM (2015). https://doi.org/10.1145/2806416.2806434
5. Chapman, C., Stolee, K.T.: Exploring regular expression usage and context in python. In: Zeller, A., Roychoudhury, A. (eds.) Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016. pp. 282–293. ACM (2016). https://doi.org/10.1145/2931037.2931073, https://doi.org/10.1145/2931037.2931073
6. contributors, W.: Regular expression—wikipedia (2019), https://en.wikipedia.org/w/index.php?title=Regular_expression&%20oldid=852858998
7. Davis, J.C.: Rethinking regex engines to address ReDoS. In: ESEC/FSE'19. pp. 1256–1258. ACM (2019)
8. Davis, J.C., Coghlan, C.A., Servant, F., Lee, D.: The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. In: Leavens, G.T., Garcia, A., Pasareanu, C.S. (eds.) Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018. pp. 246–256. ACM (2018). https://doi.org/10.1145/3236024.3236027, https://doi.org/10.1145/3236024.3236027
9. Davis, J.C., Coghlan, C.A., Servant, F., Lee, D.: The impact of regular expression denial of service (ReDoS) in practice: An empirical study at the ecosystem scale. In: ESEC/FSE'18. pp. 246–256. ACM (2018)
10. Davis, J.C., Michael IV, L.G., Coghlan, C.A., Servant, F., Lee, D.: Why aren't regular expressions a lingua franca? An empirical study on the re-use and portability of regular expressions. In: ESEC/FSE'19. pp. 1256–1258. ACM (2019)
11. Davis, J.C., Servant, F., Lee, D.: Using selective memoization to defeat regular expression denial of service (ReDoS). In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. pp. 1–17. IEEE (2021). https://doi.org/10.1109/SP40001.2021.00032, https://doi.org/10.1109/SP40001.2021.00032
12. docs.rs: regex - rust. https://docs.rs/regex/1.5.4/regex/ (2021)
13. Exchange, S.: Outage postmortem. http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016 (2016)
14. Gelade, W., Gyssens, M., Martens, W.: Regular expressions with counting: Weak versus strong determinism. In: Mathematical Foundations of Computer Science 2009. pp. 369–381. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03816-7_32
15. Gelade, W., Gyssens, M., Martens, W.: Regular expressions with counting: Weak versus strong determinism. SIAM J. Comput. **41**(1), 160–190 (2012). https://doi.org/10.1137/100814196, extended version of paper in MFCS'09
16. Glushkov, V.M.: The abstract theory of automata. Russian Math. Surveys **16**, 1–53 (1961). https://doi.org/10.1070/RM1961v016n05ABEH004112
17. Google: RE2. https://github.com/google/re2
18. Graham-Cumming, J.: Details of the Cloudflare outage on july 2, 2019. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/ (2019)
19. Haertel, M., et al.: GNU grep. https://www.gnu.org/software/grep/

20. Holík, L., Lengál, O., Saarikivi, O., Turoňová, L., Veanes, M., Vojnar, T.: Succinct determinisation of counting automata via sphere construction. In: Proc. of APLAS'19. LNCS, vol. 11893, pp. 468–489. Springer (2019). https://doi.org/10.1007/978-3-030-34175-6_24
21. Holík, L., Síč, J., Turoňová, L., Vojnar, T.: Fast matching of regular patterns with synchronizing counting (technical report). Tech. rep., Brno University of Technology (2023), https://doi.org/10.48550/arXiv.2301.12851
22. Hovland, D.: Regular expressions with numerical constraints and automata with counters. In: ICTAC. LNCS, vol. 5684, pp. 231–245. Springer (2009). https://doi.org/10.1007/978-3-642-03466-4_15
23. Hovland, D.: The membership problem for regular expressions with unordered concatenation and numerical constraints. In: Language and Automata Theory and Applications. pp. 313–324. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28332-1_27
24. Hromkovič, J., Seibert, S., Wilke, T.: Translating regular expressions into small ε-free nondeterministic finite automata. In: Reischuk, R., Morvan, M. (eds.) STACS 97. pp. 55–66. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
25. Kilpeläinen, P., Tuhkanen, R.: Regular expressions with numerical occurrence indicators - preliminary results. In: SPLST'03. pp. 163–173. University of Kuopio, Department of Computer Science (2003)
26. Kilpeläinen, P., Tuhkanen, R.: One-unambiguity of regular expressions with numeric occurrence indicators. Information and Computation **205**(6), 890–916 (2007). https://doi.org/10.1016/j.ic.2006.12.003
27. M. Roesch et al.: Snort: A Network Intrusion Detection and Prevention System,. http://www.snort.org
28. RegExLib.com: The Internet's first Regular Expression Library. http://regexlib.com/
29. Robin Sommer et al.: The Bro Network Security Monitor, http://www.bro.org
30. Saarikivi, O., Veanes, M., Wan, T., Xu, E.: Symbolic regex matcher. In: Vojnar, T., Zhang, L. (eds.) TACAS'2019. LNCS, vol. 11427, pp. 372–378. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_24, https://doi.org/10.1007/978-3-030-17462-0_24
31. Smith, R., Estan, C., Jha, S.: XFA: faster signature matching with extended automata. In: IEEE Symposium on Security and Privacy. IEEE (2008). https://doi.org/10.1109/SP.2008.14
32. Smith, R., Estan, C., Jha, S., Siahaan, I.: Fast signature matching using extended finite automaton (XFA). In: ICISS'08. LNCS, vol. 5352, pp. 158–172. Springer (2008). https://doi.org/10.1007/978-3-540-89862-7_15
33. Sperberg-McQueen, M.: Notes on finite state automata with counters. https://www.w3.org/XML/2004/05/msm-cfa.html, https://www.w3.org/XML/2004/05/msm-cfa.html, accessed: 2018-08-08
34. The Sagan team: The Sagan Log Analysis Engine, https://quadrantsec.com/sagan_log_analysis_engine/
35. Thompson, K.: Programming techniques: Regular expression search algorithm. Commun. ACM **11**(6), 419–422 (1968)
36. Turoňová, L., Holík, L., Lengál, O., Saarikivi, O., Veanes, M., Vojnar, T.: Regex matching with counting-set automata. Proc. ACM Program. Lang. **4**(OOPSLA), 218:1–218:30 (2020)
37. Turoňová, L., Holík, L., Lengál, O., Veanes, M., Vojnar, T.: Counting in regexes considered harmful (2022)
38. Češka, M., Havlena, V., Holík, L., Lengál, O., Vojnar, T.: Approximate reduction of finite automata for high-speed network intrusion detection. In: Proc. of TACAS'18. LNCS, vol. 10806. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_9

39. Wang, P., Stolee, K.T.: How well are regular expressions tested in the wild? In: Leavens, G.T., Garcia, A., Pasareanu, C.S. (eds.) Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018. pp. 668–678. ACM (2018). https://doi.org/10.1145/3236024.3236072, https://doi.org/10.1145/3236024.3236072
40. Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G., Hu, J., Zhu, H.: Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). pp. 631–648. USENIX Association, Boston, MA (Feb 2019), https://www.usenix.org/conference/nsdi19/presentation/wang-xiang
41. Wübbeling, M.: Regular expression security. ADMIN **55** (2020)
42. Yang, L., Karim, R., Ganapathy, V., Smith, R.: Improving NFA-based signature matching using ordered binary decision diagrams. In: Recent Advances in Intrusion Detection. pp. 58–78. Springer Berlin Heidelberg (2010)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.