Regular Expression Matching using Bit Vector Automata

ALEXIS LE GLAUNEC, Rice University, USA
LINGKUN KONG, Rice University, USA
KONSTANTINOS MAMOURAS, Rice University, USA

Regular expressions (regexes) are ubiquitous in modern software. There is a variety of implementation techniques for regex matching, which can be roughly categorized as (1) relying on backtracking search, or (2) being based on finite-state automata. The implementations that use backtracking are often chosen due to their ability to support advanced pattern-matching constructs. Unfortunately, they are known to suffer from severe performance problems. For some regular expressions, the running time for matching can be exponential in the size of the input text. In order to provide stronger guarantees of matching efficiency, automata-based regex matching is the preferred choice. However, even these regex engines may exhibit severe performance degradation for some patterns. The main reason for this is that regexes used in practice are not exclusively built from the classical regular constructs, i.e., concatenation, nondeterministic choice and Kleene's star. They involve additional constructs that provide succinctness and convenience of expression. The most common such construct is bounded repetition (also called counting), which describes the repetition of the pattern a fixed number of times.

In this paper, we propose a new algorithm for the efficient matching of regular expressions that involve bounded repetition. Our algorithms are based on a new model of automata, which we call nondeterministic bit vector automata (NBVA). This model is chosen to be expressively equivalent to nondeterministic counter automata with bounded counters, a very natural model for expressing patterns with bounded repetition. We show that there is a class of regular expressions with bounded repetition that can be matched in time that is independent from the repetition bounds. Our algorithms are general enough to cover the vast majority of challenging bounded repetitions that arise in practice. We provide an implementation of our approach in a regex engine, which we call BVA-Scan. We compare BVA-Scan against state-of-the-art regex engines on several real datasets.

CCS Concepts: • **Theory of computation → Formal languages and automata theory**; • **Software and its engineering**;

Additional Key Words and Phrases: regex, automata theory, bounded repetition, counter automata

ACM Reference Format:
Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2023. Regular Expression Matching using Bit Vector Automata. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 92 (April 2023), 30 pages. https://doi.org/10.1145/3586044

1 INTRODUCTION

Regular patterns, whether they are expressed using regular expressions or finite-state automata, are ubiquitous in numerous application domains. They are used in text analysis for searching, extracting, parsing or transforming pieces of text [Aho and Corasick 1975; Awk 2022; Grep 2022]. They are also used in network security [Yu et al. 2006] for describing signatures in network traffic

Authors' addresses: Alexis Le Glaunec, Department of Computer Science, Rice University, USA, alexis.leglaunec@rice.edu; Lingkun Kong, Department of Computer Science, Rice University, USA, klk@rice.edu; Konstantinos Mamouras, Department of Computer Science, Rice University, USA, mamouras@rice.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).

2475-1421/2023/4-ART92
https://doi.org/10.1145/3586044

that indicate an intrusion or other security issues, in bioinformatics [Bo et al. 2018; Roy and Aluru 2016] for specifying DNA, RNA or protein sequences, and in runtime verification [Barringer et al. 2004; Bartocci et al. 2018] for describing safety properties. The usefulness of regular patterns is also witnessed by the widespread use of regexes in software projects. It is reported in several studies (see, e.g., [Davis 2019]) that 30%-40% of Java, JavaScript and Python software projects use regex matching.

There are several approaches for implementing regular pattern matching. The regex matching engines of many programming languages such as Java, .NET, Python, and JavaScript use backtracking search. For some regexes, these engines need time that is exponential in the size of the input text. For other regexes, backtracking engines may need time that is polynomial in the length of the input text, but the polynomial has degree at least 2. This is a lot worse than Thompson's algorithm [Thompson 1968], which has time complexity $O(m \cdot n)$, where m is the size of the regular expression and n is the size of the input text. The benefit of backtracking is the simplicity of its implementation and the ease with which advanced features (e.g., lookaround and backreferences) can be added.

Many modern regex engines are based on the classical theory of automata. They employ DFAs or NFAs, or both [Grep 2022; RE2 2023; Wang et al. 2019]. DFA-based algorithms perform a memory lookup when receiving an input symbol to execute the transition of the automaton. This makes them fast, because they need $O(1)$ time per symbol. One potential problem is that the representation of the DFA might require a large amount of memory. For many patterns that arise in practice, their encoding as NFAs can be substantially more compact that their encoding as DFAs. In fact, it is known that there are families of patterns for which minimal DFAs are exponentially larger than equivalent NFAs [Meyer and Fischer 1971]. The downside with NFAs is that their execution is less efficient in terms of time complexity. NFA execution involves maintaining a set of active states, which represent several possible execution paths, and performing at every step a transition for each one of the currently active states. So, for every step, this may involve work that is proportional to the number of states of the NFA.

Software-based (i.e., CPU-based) implementations of regular pattern matching often explore this trade-off between time- and memory-efficiency that we described in the previous paragraph. There are also hardware-based implementations which make use of the inherent parallelism that is available in hardware, where circuit elements compute independently and in parallel. There are several proposals that use field-programmable gate arrays (FPGAs) [Bispo et al. 2006] and others that employ digital application-specific integrated circuit (ASIC) accelerators [Brodie et al. 2006]. The latest hardware technology is the in-memory architecture, which is based on NFAs and processes the NFA transitions directly inside memories. This offers massive parallelism by combining memory and computation at the same physical location on the circuit. The Automata Processor (AP) from Micron [Dlugosch et al. 2014] is the first example of such an architecture. Hardware offers the memory-efficiency of NFAs without a penalty in execution time, since NFA transitions from active states are explored in parallel in one step. In recent work, [Kong et al. 2022] propose an architecture based on nondeterministic counter automata (NCAs) instead of NFAs.

While an explicit NFA representation can be much more compact than an explicit DFA representation, there are patterns for which even an NFA is too large. Such patterns are very common in practice, because regular patterns are typically specified using an extended regex syntax that makes them highly succinct. Classical regular expressions include constructs for concatenation, nondeterministic choice and Kleene's star. A regular expression with these constructs can be translated into an NFA whose size is linear in the size of the expression [Antimirov 1996; Glushkov 1961; Thompson 1968]. In practice, regular expressions also include the constructs of (1) *bounded repetition* (also called *counting*), which is written as $r\{m, n\}$, (2) and *lookaround*, which includes positive/negative lookahead, and positive/negative lookbehind. Even though lookaround is a regular construct, it is not supported by any of the available automata-based regex engines (e.g., grep, RE2, Hyperscan). For this reason, this paper focuses on bounded repetition. Bounded repetition is extremely common in practical use cases of regular expressions. For example, the majority of the regexes that are found in datasets of intrusion detection rules [Snort 2023; Suricata 2023] include bounded repetition. Regexes similar to `.*a.{`n`}` arise in these datasets. It is known that the smallest NFAs that recognize these regexes have $\Theta(n)$ states, and the minimal DFAs that recognize them have $\Theta(2^n)$ states [Meyer and Fischer 1971]. The regex `.*a.{`n`}` has size $\Theta(\log n)$ because the repetition bound n is represented in binary/decimal notation. So, any NFA that recognizes this pattern is of size exponential in the regex size.

The exponential succinctness of bounded repetition can have a large impact on the performance of software regex engines (grep, RE2) that generate and cache the pattern DFA on the fly. When the DFA becomes too large, these engines start flushing the DFA cache and eventually revert to a much slower NFA simulation. While this problem is well understood, there are no known ways to mitigate it in the general case. The recent paper [Turoňová et al. 2020] explores an efficient algorithm for matching regexes with bounded repetition, but only addresses patterns of a limited form. For example, the patterns `.*(aa){100}` and `.*a{0,100}a{200}` are not handled by the algorithm of [Turoňová et al. 2020]. More information about [Turoňová et al. 2020] can be found in Section 6.

The present paper improves upon this situation by substantially expanding the class of regular expressions with bounded repetition that can be matched efficiently. In particular, our main contributions are the following:

(1) We propose a new efficient algorithm for matching every bounded repetition of the form

$$(\sigma_0\sigma_1\ldots\sigma_{k-1})\{m,n\},$$

where each σ_i is a predicate (i.e., a character class). This algorithm applies to bounded repetitions (of this particular form) that can occur as subexpressions of a larger regular expression. The time complexity of our algorithm is $O(k)$ per step. Our algorithm is based on a model of automata which we call nondeterministic bit vector automata (NBVAs). These automata have a precise correspondence to the classical model of nondeterministic counter automata (NCAs) with bounded counters.

(2) We extend the previous result to the larger class of bounded repetitions of the form $r\{m,n\}$, where r is a regex without bounded repetitions that recognizes strings of the same length. The time complexity of our algorithm is $O(\ell)$ per step, where ℓ is the number of predicate (i.e., character class) occurrences in r.

(3) We provide an implementation of a general regex engine, which we call ***BVA-Scan***, that supports bounded repetition and integrates our efficient algorithms. We compare our regex engine against state-of-the-art regex engines (RE2 [RE2 2023], PCRE [Hazel and Herczeg 2022], and CA [Library 2021]) using the real application benchmarks Snort [Roesch 1999; Snort 2023], Suricata [Suricata 2023], Protomata [Roy and Aluru 2016; Sigrist et al. 2009], ClamAV [ClamAV 2023], SpamAssassin [SpamAssassin 2022], and RegexLib [RegexLib 2023]. Our results show that BVA-Scan consistently outperforms all three regex engines we are comparing against. The outperformance of our engine is substantial for cases of computationally challenging bounded repetition.

Outline of paper: In Section 2, we provide the definitions of regular expressions with bounded repetition and nondeterministic counter automata (NCAs) that we will use throughout the paper. Section 3 introduces the model of nondeterministic bit vector automata (NBVAs) as an alternative to NCAs that is more convenient for specifying regex matching algorithms. In Section 4, we present our algorithms for the efficient matching of a form of bounded repetition that arises frequently in practice. Section 5 presents our experimental results against three state-of-the-art regex engines over several real benchmarks. Section 6 contains a discussion of related work. Finally, we conclude in Section 7 with a brief summary of our contributions.

2 PRELIMINARIES

In this section, we present a variant of nondeterministic counter automata (NCAs) that is appropriate for implementing regular expressions with bounded repetition. We are not interested in NCAs with unbounded counters, which can recognize non-regular languages. Instead, we focus on NCAs with bounded counters, which are finite-state machines and can therefore only recognize regular languages. Most definitions of NCAs in prior works consider the same set of counter registers for each control state of the automaton. We relax this restriction by allowing each state to have a different number of counters. This flexibility is useful for reducing the amount of memory needed for NCA execution.

Let Σ be a finite alphabet. We define the set Regex of ***regular expressions*** (or *regexes*) over Σ to be the smallest set that satisfies the following properties:

(1) Regex contains the expression ε (regex that recognizes the empty string).
(2) Regex contains every predicate σ over the alphabet (i.e., $\sigma \subseteq \Sigma$).
(3) For every $r, r_1, r_2 \in$ Regex, we have that $r_1 \cdot r_2 \in$ Regex (concatenation), $r_1 | r_2 \in$ Regex (nondeterministic choice) and $r^* \in$ Regex (Kleene's star).
(4) For all integers m, n with $0 \leq m \leq n$ and every $r \in$ Regex, we have that $r\{m, n\} \in$ Regex.

The concatenation symbol is sometimes omitted, i.e., we write $r_1 r_2$ instead of $r_1 \cdot r_2$. An expression of the form $r\{m, n\}$ is called a *bounded repetition*, because it describes the repetition of r from m to n times. We write $r\{n\}$ as abbreviation for $r\{n, n\}$. The expression $r\{n,\} = r\{n\}r^*$ describes the repetition of r at least n times. We say that a regular expression is *counting-free* if it does not contain any occurrence $r\{m, n\}$ of bounded repetition. The *interpretation* of a regex r is a language $\mathcal{L}(r) \subseteq \Sigma^*$, which is defined in the standard way.

Notation for character classes: A predicate over the alphabet is also called a *character class*. The predicate Σ contains all symbols in the alphabet. When we use a symbol $a \in \Sigma$ in a regex, it should be understood as the singleton predicate $\{a\} \subseteq \Sigma$. The notation $[a_1 \ldots a_n]$ is used for representing the predicate $\{a_1, \ldots, a_n\} \subseteq \Sigma$. We write $[\hat{\ } a_1 \ldots a_n]$ for the predicate $\Sigma \setminus \{a_1, \ldots, a_n\}$ that contains all symbols except for $a_1, \ldots, a_n$.

In order to define counter automata, we first fix an infinite set *CReg* of *counter registers* or, simply, *counters*. We typically write $x, y, z, \ldots$ to denote counter registers. For a subset $V \subseteq \mathit{CReg}$ of counters, a function $\beta : V \to \mathbb{N}$, which assigns a value (more specifically, a natural number) to each counter in V, is called a *V-valuation*.

*Definition 2.1 (**Counter Automata**).* Let Σ be a finite alphabet. A *nondeterministic counter automaton (NCA)* with input alphabet Σ is a tuple $\mathcal{A} = (Q, R, \Delta, I, F)$, where

- Q is a finite set of *(control) states*,
- $R : Q \to \mathcal{P}(\mathit{CReg})$ is a function that maps each state to a finite set of counters,
- Δ is the *transition relation*, which contains finitely many transitions of the form $(p, \sigma, \varphi, q, \vartheta)$, where p is the source state, $\sigma \subseteq \Sigma$ is a predicate over the alphabet, $\varphi \subseteq (R(p) \to \mathbb{N})$ is a predicate over $R(p)$-valuations, q is the destination state, and $\vartheta : (R(p) \to \mathbb{N}) \to (R(q) \to \mathbb{N})$,
- I is the *initialization function*, which specifies the set of *initial valuations* $I(q) \subseteq R(q) \to \mathbb{N}$ for each state q, and
- F is the *finalization function*, which specifies a set of *final valuations* $F(q) \subseteq R(q) \to \mathbb{N}$ for each state q.

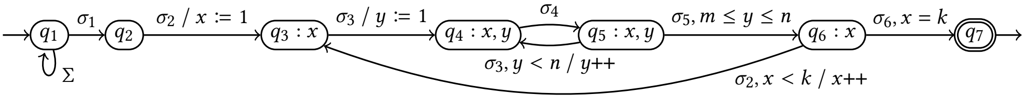


Fig. 1. NCA with two counters (x and y) for the regex $\Sigma^*\sigma_1(\sigma_2(\sigma_3\sigma_4)\{m,n\}\sigma_5)\{k\}\sigma_6$ with $1 \le m \le n$ and $k \ge 1$. The states q_1, q_2, q_7 have no counter (i.e., they are pure). The states q_3, q_6 have 1 counter (x) and the states q_4, q_5 have 2 counters (x and y).

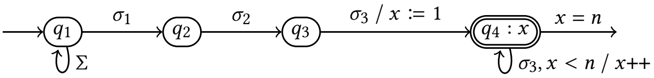
We say that a state $q \in Q$ is *pure* if $R(q) = \emptyset$, that is, it has no counter associated with it. If $R(q) \neq \emptyset$, then we say that q is a *counting* state. A state $q \in Q$ is called *final* if $F(q) \neq \emptyset$. Similarly, a state $q \in Q$ is said to be *initial* if $I(q) \neq \emptyset$.

Our definition for NCAs is similar to the one given in [Kong et al. 2022]. One difference is that we consider here a slightly more general initialization function that allows several different initial valuations for an initial state.

According to Definition 2.1, the states in an NCA do not all necessarily have the same number of counters. In fact, some states may not have any counter at all. In a transition $(p, \sigma, \varphi, q, \vartheta)$, we will call the predicate φ a *guard* because it may restrict a transition based on the values of the counters. The function ϑ of a transition is called an *action*, because it describes how to assign counter values in the destination state given the counter values in the source state.

We use a variant of the Glushkov construction [Gelade et al. 2009; Glushkov 1961; Hovland 2009] to convert a regular expression r (with bounded repetition) to an NCA that recognizes the language $\mathcal{L}(r)$. In contrast to Thompson's construction [Thompson 1968], Glushkov's construction results in ε-free automata that are also *homogeneous*, i.e., all incoming transitions of a state are labeled with the same predicate over the alphabet.

Example 2.2. Consider the regular expression $r_1 = \Sigma^*\sigma_1\sigma_2\sigma_3\{n\}$ with $n \ge 1$, where $\sigma_1, \sigma_2, \sigma_3$ are predicates over the alphabet Σ[1]. The following automaton recognizes the language of r_1:



The automaton shown above has four states: q_1, q_2, q_3, and q_4. We write $q_4 : x$ to indicate that $R(q_4) = \{x\}$. That is, q_4 has one counter register called x. Notice that q_1 has no annotation with counters, which means that $R(q_1) = \emptyset$ (i.e., q_1 is pure). Each edge $p \longrightarrow q$ is annotated with an expression of the form $\sigma, \varphi \,/\, \vartheta$, where

(1) σ is a predicate over Σ (i.e., a character class),
(2) φ is a guard over the counters of p, and
(3) ϑ is an assignment that specifies values for the counters of q using the values for counters of p.

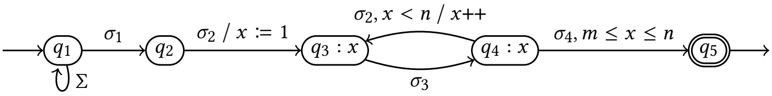
Here are some notational conventions that we follow when drawing diagrams of NCAs:

- When we omit the guard φ, then this indicates that it is always true. More formally, if an edge emanating from a state p is not annotated with a guard φ, then it should be understood that $\varphi = (R(p) \to \mathbb{N})$.
- For an edge from p to q, we sometimes omit the action ϑ. We only do this when $R(q) \subseteq R(p)$, and the omission of the action ϑ indicates that the counters $R(q)$ retain the values from the

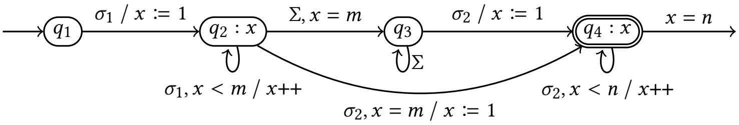
[1] In order to make the regular expression more concrete, suppose that $\sigma_1 = [ab]$, $\sigma_2 = [bcd]$, and $\sigma_2 = [\hat{}a]$. So, the regular expression r_1 is the same as .* `[ab][bcd][^a]{`n`}` using POSIX notation [in PCRE 2022]. The expression Σ^* has the same meaning as .* in POSIX notation.

previous state. Alternatively, we can explicitly indicate this action that does not change counter values by writing one or more assignments of the form "$x := x$".
- We write "$x = n$" (resp., "$x < n$") for the guard that checks whether the value of counter x is equal to n (resp., strictly less than n).
- We write "$x := c$" to denote the assignment (action) of the value c to the counter x.
- The action "x++" indicates the incrementation of the counter x by 1. So, it can be understood as an abbreviation for the assignment "$x := x + 1$".
- We use double circle notation to indicate that a state is final (see state q_4 in the diagram shown earlier). An arrow emanating from a final state q is annotated with the predicate $F(q)$ over counter valuations (recall that F is the finalization function).
- If $F(q) = (R(q) \to \mathbb{N})$, then we may omit the finalization function from the diagram.

The examples that follow (for regexes r_2, r_3, r_4) are adapted from [Kong et al. 2022]. The following automaton recognizes the language of the regex $r_2 = \Sigma^*\sigma_1(\sigma_2\sigma_3)\{m, n\}\sigma_4$, where σ_1, σ_2, σ_3, σ_4 are character classes and $1 \leq m \leq n$.



Observe in the diagram above the omission of both the guard and the action from the edge $q_3 \to q_4$. As mentioned earlier, this indicates that the guard is always true and the action is the assignment "$x := x$" that copies the counter value. The language of the regex $r_3 = \sigma_1\{m\}\Sigma^*\sigma_2\{n\}$, where $m, n \geq 1$ and σ_1, σ_2 are character classes, is recognized by the following automaton:



All automata that we have seen so far use at most one counter for each control state. For the regex $r_4 = \Sigma^*\sigma_1(\sigma_2(\sigma_3\sigma_4)\{m, n\}\sigma_5)\{k\}\sigma_6$ with $1 \leq m \leq n$ and $k \geq 1$ we need two counters for some states because of the nesting of bounded repetition. See Fig. 1.

Nondeterministic semantics. We will now consider a nondeterministic path semantics for NCAs, which is analogous to the path semantics for NFAs. Let $\mathcal{A} = (Q, R, \Delta, I, F)$ be an NCA.

A *token* for $\mathcal{A}$ is a pair (q, β), where q is a state and $\beta : R(q) \to \mathbb{N}$ is a counter valuation for q. The set of all tokens for $\mathcal{A}$ is denoted by $\mathbf{Tk}(\mathcal{A})$. That is,

$$\mathbf{Tk}(\mathcal{A}) = \{(q, \beta) \mid q \in Q \text{ and } \beta : R(q) \to \mathbb{N}\}.$$

Simplified notation for tokens: In many cases, we only consider states that have at most one counter register. We simplify the notation for tokens as described below:
- For a pure state q (i.e., a state with no counter, see Definition 2.1), there is only one possible valuation, denoted $0_{\mathbb{N}} : \emptyset \to \mathbb{N}$, which carries no information. That is, $R(q) = \emptyset$ and therefore the set of valuations $R(q) \to \mathbb{N}$ is equal to $\{0_{\mathbb{N}}\}$. So, we will often abuse notation and simply write q for the token $(q, 0_{\mathbb{N}})$.
- Let us consider now a state q with one counter, i.e., $R(q) = \{x\}$ for some $x \in \mathit{CReg}$. In this case, a valuation β (of type $\{x\} \to \mathbb{N}$) for q specifies only one value $c = \beta(x) \in \mathbb{N}$ for the unique counter x for q. For this reason, we will sometimes write (q, c) instead of (q, β).

For a letter $a \in \Sigma$, we define the *token transition relation* $\to^a$ on $\mathbf{Tk}(\mathcal{A})$ as follows: $(p, \beta) \to^a (q, \gamma)$ iff there is a transition $(p, \sigma, \varphi, q, \vartheta) \in \Delta$ with $a \in \sigma$ such that $\beta \in \varphi$ and $\gamma = \vartheta(\beta)$. A token (q, β) is *initial* if $\beta \in I(q)$ (which, in particular, means that q is initial). A token (q, β) is *final* if $\beta \in F(q)$ (which also means that q is final). A *run* of $\mathcal{A}$ on an input string $a_1 a_2 \dots a_n \in \Sigma^*$ is a sequence

$$(q_0, \beta_0) \xrightarrow{a_1} (q_1, \beta_1) \xrightarrow{a_2} (q_2, \beta_2) \xrightarrow{a_3} \cdots \xrightarrow{a_n} (q_n, \beta_n),$$

where each (q_i, β_i) is a token, (q_0, β_0) is an initial token, and $(q_{i-1}, \beta_{i-1}) \to^a (q_i, \beta_i)$ for every $i = 1, \dots, n$. A run is *accepting* if it ends with a final token. The NCA $\mathcal{A}$ *accepts* a string if there is an accepting run on it. We denote by $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ the set of all strings that are accepted by $\mathcal{A}$.

For an NCA $\mathcal{A}$, the set of tokens $\mathbf{Tk}(\mathcal{A})$ together with the transition relations $\to^a$ forms a labeled transition system. The family of transition relations $(\to^a)_{a \in \Sigma}$ can be represented as a ternary relation $\to \subseteq \mathbf{Tk}(\mathcal{A}) \times \Sigma \times \mathbf{Tk}(\mathcal{A})$.

Deterministic semantics using configurations. Let $\mathcal{A} = (Q, R, \Delta, I, F)$ be an NCA. A *configuration* S for $\mathcal{A}$ is a set of tokens for $\mathcal{A}$, that is, $S \subseteq \mathbf{Tk}(\mathcal{A})$. We write $\mathbf{C}(\mathcal{A})$ for the set of all configurations for $\mathcal{A}$. The configuration transition function $\delta : \mathbf{C}(\mathcal{A}) \times \Sigma \to \mathbf{C}(\mathcal{A})$ is defined by

$$\delta(S, a) = \{(q, \gamma) \mid (p, \beta) \to^a (q, \gamma) \text{ for some } (p, \beta) \in S\}.$$

The transition function is extended to $\delta : \mathbf{C}(\mathcal{A}) \times \Sigma^* \to \mathbf{C}(\mathcal{A})$ as follows: $\delta(S, \varepsilon) = S$ and $\delta(S, xa) = \delta(\delta(S, x), a)$ for every $S \subseteq \mathbf{Tk}(\mathcal{A})$, $x \in \Sigma^*$ and $a \in \Sigma$. The *initial configuration* S_0 is the set of all initial tokens. Define the function $[\![\mathcal{A}]\!] : \Sigma^* \to \mathbf{C}(\mathcal{A})$ by $[\![\mathcal{A}]\!](x) = \delta(S_0, x)$. Intuitively, $[\![\mathcal{A}]\!](x)$ is the configuration of $\mathcal{A}$ after consuming the string x. This semantics coincides with $\mathcal{L}(\mathcal{A})$ in the following sense: for every $x \in \Sigma^*$, $x \in \mathcal{L}(\mathcal{A})$ iff $[\![\mathcal{A}]\!](x)$ contains some final token.

Bounded counters: NCAs with unbounded counters can recognize non-regular languages. For this reason, we will restrict them so that there is a fixed bound on all counter values.

Let $\mathcal{A}$ be an NCA and $n \in \mathbb{N}$. A token (q, β) is said to be *n-bounded* if $\beta(x) \leq n$ for every counter $x \in R(q)$. We also say that $\mathcal{A}$ (resp., a state q) is *n-bounded* if every token (resp., token on state q) reachable from some initial token is n-bounded. Finally, the NCA $\mathcal{A}$ is said to *have bounded counters* if there exists some $n \in \mathbb{N}$ such that $\mathcal{A}$ is n-bounded. NCAs with bounded counters have the same expressiveness as finite-state automata (i.e., DFAs and NFAs), but they may be more succinct. See [Meyer and Stockmeyer 1972] and [Stockmeyer and Meyer 1973] for some implications of succinctness due to counting.

The automata that we consider are constructed from regexes (with bounded repetition) using the Glushkov construction. Every action of the form x++ (counter incrementation) is guarded by a test $x < n$ because it corresponds to a subexpression of the form $r\{m, n\}$. So, every automaton that is constructed from regexes has bounded counters. Moreover, for every control state and every counter, we can easily obtain an upper bound for the counter values by inspecting the automaton. For example, let us consider the NCA of Fig. 1. Counter x at states q_3, q_4, q_5, q_6 is bounded above by k because $(q_6, \sigma_2, \text{“}x < k\text{”}, q_3, \text{“}x\text{++”})$ is the only transition that increments x. The transition guard will not allow the value of the counter to exceed k. Similarly, counter y at states q_4, q_5 is bounded above by n because $(q_5, \sigma_3, \text{“}y < n\text{”}, q_4, \text{“}y\text{++”})$ is the only transition that increments y.

If we restrict counter values according to the bounds described earlier, then we have a finite number of counter valuations that can be associated with each control state. Under this restriction, the set $\mathbf{Tk}(\mathcal{A})$ of tokens is finite and it can be viewed as the state space of an NFA that recognizes the same language as the NCA $\mathcal{A}$.

3 BIT VECTOR AUTOMATA

In this section, we will introduce the model of nondeterministic bit vector automata (NBVAs). These automata are expressively equivalent to the NCAs of Section 2, assuming that the NCA counters are bounded. NBVAs are more convenient, however, for formulating pattern matching algorithms. Informally, if $\mathcal{A}$ is an NCA, then the corresponding NBVA $\mathcal{A}'$ differs from $\mathcal{A}$ in how its configuration is represented. The configuration of the NBVA $\mathcal{A}'$ specifies for each control state q a data structure (more specifically, a bit vector) to represent the set of active tokens (for the NCA $\mathcal{A}$) of the form (q, β), i.e., tokens that are on the control state q.

Let $\mathcal{A} = (Q, R, \Delta, I, F)$ be an NCA. For a state $q \in Q$ and a subset $T \subseteq \mathbf{Tk}(\mathcal{A})$ of tokens for the automaton, define $T|_q = T \cap (\{q\} \times (R(q) \to \mathbb{N}))$. That is, $T|_q$ contains those tokens of T whose first component is the state q. The operational intuition is that $[\![\mathcal{A}]\!](x)|_q$ is the set of tokens that we get on state q when we execute the NCA $\mathcal{A}$ on input x.

*Definition 3.1 (**Nondeterministic Bit Vector Automaton**).* Let Σ be a finite alphabet. A *nondeterministic bit vector automaton* (NBVA) is a tuple (Q, w, Δ, I, F), where

- Q is a finite set of *(control) states*,
- $w : Q \to \{1, 2, \ldots\}$ is a function that maps each state to a strictly positive integer,
- Δ is the *transition relation*, which contains finitely many transitions of the form $(p, \sigma, q, \vartheta)$, where p is the source state, $\sigma \subseteq \Sigma$ is a predicate over the alphabet, q is the destination state, and $\vartheta : \mathbb{B}^{w(p)} \to \mathbb{B}^{w(q)}$,
- I is the *initialization function*, which specifies an *initial vector* $I(q) : \mathbb{B}^{w(q)}$ for each state q, and
- F is the *finalization function*, which specifies a function $F(q) : \mathbb{B}^{w(q)} \to \mathbb{B}$ for each state q.

A state q is *initial* if $I(q) \neq \mathbf{0}_{w(q)}$, where $\mathbf{0}_{w(q)}$ is the zero vector of length $w(q)$. A state q is *final* if $F(q)(v) = 1$ for some $v \in \mathbb{B}^{w(q)}$. A state $q \in Q$ is *pure* (resp., *counting*) if $w(q) = 1$ (resp., $w(q) > 1$).

In contrast to the NCAs of Definition 2.1, the NBVAs of Definition 3.1 do not have transition guards (i.e., predicates over the bit vectors). We have chosen this presentation because guards are redundant. If we had a separate guard $\varphi \subseteq \mathbb{B}^{w(p)}$ and an action $\vartheta : \mathbb{B}^{w(p)} \to \mathbb{B}^{w(q)}$, then we could integrate both of them into the action $\vartheta_\varphi : \mathbb{B}^{w(p)} \to \mathbb{B}^{w(q)}$, given by

$$\vartheta_\varphi(v) = \vartheta(v), \text{ if } v \in \varphi \qquad\qquad \vartheta_\varphi(v) = \mathbf{0}, \text{ if } v \notin \varphi$$

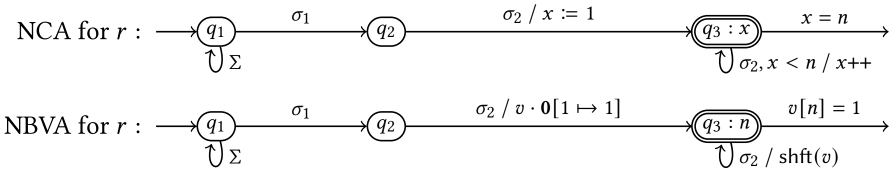
for every bit vector $v \in \mathbb{B}^{w(p)}$.

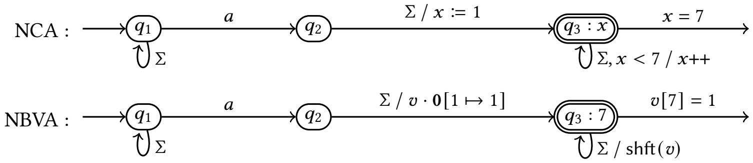
Semantics of NBVAs. Let $\mathcal{A} = (Q, w, \Delta, I, F)$ be an NBVA. A *configuration* for $\mathcal{A}$ is a mapping C, which specifies a bit vector $C(q) \in \mathbb{B}^{w(q)}$ for every state $q \in Q$. Intuitively, a configuration specifies the bit vector that each control state carries. We write $\mathbf{C}(\mathcal{A})$ to denote the set of all configurations of $\mathcal{A}$. We will define now the function $[\![\mathcal{A}]\!] : \Sigma^* \to \mathbf{C}(\mathcal{A})$, where $[\![\mathcal{A}]\!](x)$ is the configuration of the automaton after consuming an input string $x \in \Sigma^*$. For the base case, we define the *initial configuration* $[\![\mathcal{A}]\!](\varepsilon)$ as follows: $[\![\mathcal{A}]\!](\varepsilon)(q) = I(q)$ for every $q \in Q$. For the step case, we define

$$[\![\mathcal{A}]\!](xa)(q) = \sum \Big\{ \vartheta([\![\mathcal{A}]\!](x)(p)) \mid (p, \sigma, q, \vartheta) \in \Delta \text{ s.t. } a \in \sigma \Big\}$$

for all $x \in \Sigma^*$, $a \in \Sigma$ and $q \in Q$. The symbol $\sum$ above is bitwise OR (disjunction). The NBVA $\mathcal{A}$ *accepts* a string $x \in \Sigma^*$ if there exists a state $q \in Q$ such that $F(q)(v) = 1$, where $v = [\![\mathcal{A}]\!](x)(q)$.

Example 3.2. Consider the regex $r = \Sigma^* \sigma_1 \sigma_2 \{n\}$ with $n \geq 1$, where σ_1, σ_2 are predicates over the alphabet. The following automata recognize the language of r:





input	NCA configuration	NBVA configuration	output
	q_1	$q_1 \mapsto 1$, $q_2 \mapsto 0$, $q_3 \mapsto \langle 0,0,0,0,0,0,0 \rangle$	0
b	q_1	$q_1 \mapsto 1$, $q_2 \mapsto 0$, $q_3 \mapsto \langle 0,0,0,0,0,0,0 \rangle$	0
a	q_1, q_2	$q_1 \mapsto 1$, $q_2 \mapsto 1$, $q_3 \mapsto \langle 0,0,0,0,0,0,0 \rangle$	0
b	$q_1, (q_3, 1)$	$q_1 \mapsto 1$, $q_2 \mapsto 0$, $q_3 \mapsto \langle 1,0,0,0,0,0,0 \rangle$	0
a	$q_1, q_2, (q_3, 2)$	$q_1 \mapsto 1$, $q_2 \mapsto 1$, $q_3 \mapsto \langle 0,1,0,0,0,0,0 \rangle$	0
a	$q_1, q_2, (q_3, 1), (q_3, 3)$	$q_1 \mapsto 1$, $q_2 \mapsto 1$, $q_3 \mapsto \langle 1,0,1,0,0,0,0 \rangle$	0
b	$q_1, (q_3, 1), (q_3, 2), (q_3, 4)$	$q_1 \mapsto 1$, $q_2 \mapsto 0$, $q_3 \mapsto \langle 1,1,0,1,0,0,0 \rangle$	0
a	$q_1, q_2, (q_3, 2), (q_3, 3), (q_3, 5)$	$q_1 \mapsto 1$, $q_2 \mapsto 1$, $q_3 \mapsto \langle 0,1,1,0,1,0,0 \rangle$	0
a	$q_1, q_2, (q_3, 1), (q_3, 3), (q_3, 4), (q_3, 6)$	$q_1 \mapsto 1$, $q_2 \mapsto 1$, $q_3 \mapsto \langle 1,0,1,1,0,1,0 \rangle$	0
a	$q_1, q_2, (q_3, 1), (q_3, 2), (q_3, 4), (q_3, 5), (q_3, 7)$	$q_1 \mapsto 1$, $q_2 \mapsto 1$, $q_3 \mapsto \langle 1,1,0,1,1,0,1 \rangle$	1
a	$q_1, q_2, (q_3, 1), (q_3, 2), (q_3, 3), (q_3, 5), (q_3, 6)$	$q_1 \mapsto 1$, $q_2 \mapsto 1$, $q_3 \mapsto \langle 1,1,1,0,1,1,0 \rangle$	0
b	$q_1, (q_3, 1), (q_3, 2), (q_3, 3), (q_3, 4), (q_3, 6), (q_3, 7)$	$q_1 \mapsto 1$, $q_2 \mapsto 0$, $q_3 \mapsto \langle 1,1,1,1,0,1,1 \rangle$	1
b	$q_1, (q_3, 2), (q_3, 3), (q_3, 4), (q_3, 5), (q_3, 7)$	$q_1 \mapsto 1$, $q_2 \mapsto 0$, $q_3 \mapsto \langle 0,1,1,1,1,0,1 \rangle$	1
a	$q_1, q_2, (q_3, 3), (q_3, 4), (q_3, 5), (q_3, 6)$	$q_1 \mapsto 1$, $q_2 \mapsto 1$, $q_3 \mapsto \langle 0,0,1,1,1,1,0 \rangle$	0
b	$q_1, (q_3, 1), (q_3, 4), (q_3, 5), (q_3, 6), (q_3, 7)$	$q_1 \mapsto 1$, $q_2 \mapsto 0$, $q_3 \mapsto \langle 1,0,0,1,1,1,1 \rangle$	1

Fig. 2. Execution of the NCA and the NBVA for the regular expression `.*a.{7}`.

Notation: We write $\mathbf{0}[1 \mapsto 1]$ to denote the bit vector that is zero everywhere, except for position 1, where it is equal to 1. This is an instance of the more general notation $v[i \mapsto b]$, where v is a bit vector, i is a position, and b is a Boolean value. The bit vector $v[i \mapsto b]$ results from v by modifying it (potentially) at position i so that it has the value b. If v is a Boolean value and w is a bit vector of size n, then $v \cdot w$ is a bit vector of size n given by $0 \cdot w = \mathbf{0}$ and $1 \cdot w = w$.

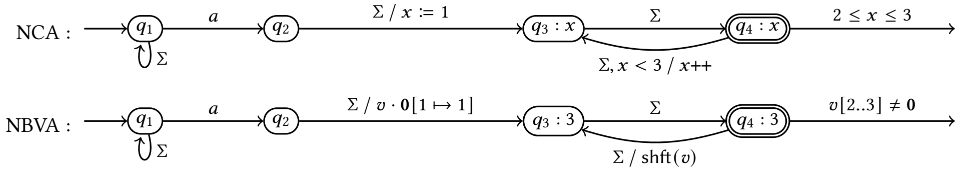
The NBVA shown earlier (for the regex $r = \Sigma^* \sigma_1 \sigma_2\{n\}$) has three states: q_1, q_2, and q_3. We write "$q_3 : n$" to indicate that $w(q_3) = n$, i.e., q_3 carries a bit vector of size n. Notice that q_1 has no annotation, which means that $w(q_1) = 1$ (i.e., q_1 is pure). We annotate each edge $p \longrightarrow q$ with an expression of the form σ / ϑ, where σ is a predicate over Σ, and ϑ is a function for computing the bit vector of q using the bit vector of p. We use v as a symbol that represents the bit vector of p. Here are some explanations on the notation we use when drawing NBVA diagrams:

- The action ϑ is omitted only when $w(p) = w(q)$. The omission of ϑ indicates that the bit vector is propagated unchanged, i.e., ϑ is the identity function. We can indicate this explicitly in an NBVA diagram by writing the expression "v" that represents the identity function.
- We write "$v[n] = 1$" to denote the function of type $\mathbb{B}^n \longrightarrow \mathbb{B}$ (which can also be viewed as a predicate over $\mathbb{B}^n$) that checks whether the value of v at the n-th position is equal to 1.
- We write "shft(v)" to denote the action (function of type $\mathbb{B}^n \longrightarrow \mathbb{B}^n$) that shifts a bit vector by one position. More formally, the shift operation is defined as follows:

$$\mathsf{shft}(v)[1] = 0 \quad \text{and} \quad \mathsf{shft}(v)[i] = v[i-1] \text{ for every } i = 2, \ldots, n.$$

We use double circle notation to indicate that a state is final (see state q_3 above). An arrow emanating from a final state q is annotated with a description of the function $F(q) : \mathbb{B}^{w(q)} \longrightarrow \mathbb{B}$.

Fig. 2 shows the execution of the NCA and NBVA for the regex $\Sigma^* a \Sigma\{7\}$ (or `.*a.{7}` in POSIX notation), which has the same form as $r = \Sigma^* \sigma_1 \sigma_2\{n\}$.



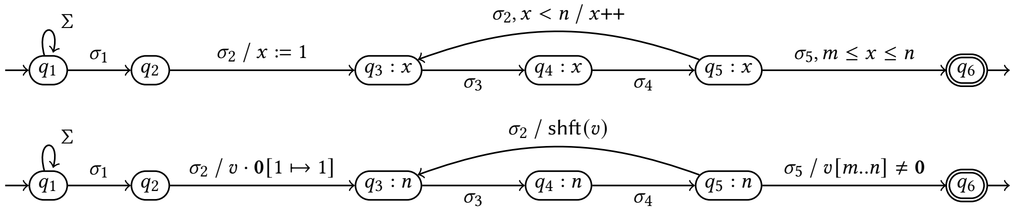
input	NCA configuration	NBVA configuration				output
	q_1	$q_1 \mapsto 1,$	$q_2 \mapsto 0,$	$q_3 \mapsto \langle 0,0,0 \rangle,$	$q_4 \mapsto \langle 0,0,0 \rangle$	0
b	q_1	$q_1 \mapsto 1,$	$q_2 \mapsto 0,$	$q_3 \mapsto \langle 0,0,0 \rangle,$	$q_4 \mapsto \langle 0,0,0 \rangle$	0
a	q_1, q_2	$q_1 \mapsto 1,$	$q_2 \mapsto 1,$	$q_3 \mapsto \langle 0,0,0 \rangle,$	$q_4 \mapsto \langle 0,0,0 \rangle$	0
a	$q_1, q_2, (q_3, 1)$	$q_1 \mapsto 1,$	$q_2 \mapsto 1,$	$q_3 \mapsto \langle 1,0,0 \rangle,$	$q_4 \mapsto \langle 0,0,0 \rangle$	0
a	$q_1, q_2, (q_3, 1), (q_4, 1)$	$q_1 \mapsto 1,$	$q_2 \mapsto 1,$	$q_3 \mapsto \langle 1,0,0 \rangle,$	$q_4 \mapsto \langle 1,0,0 \rangle$	0
b	$q_1, (q_3, 1), (q_4, 1), (q_3, 2)$	$q_1 \mapsto 1,$	$q_2 \mapsto 0,$	$q_3 \mapsto \langle 1,1,0 \rangle,$	$q_4 \mapsto \langle 1,0,0 \rangle$	0
b	$q_1, (q_4, 1), (q_3, 2), (q_4, 2)$	$q_1 \mapsto 1,$	$q_2 \mapsto 0,$	$q_3 \mapsto \langle 0,1,0 \rangle,$	$q_4 \mapsto \langle 1,1,0 \rangle$	1
a	$q_1, q_2, (q_3, 2), (q_4, 2), (q_3, 3)$	$q_1 \mapsto 1,$	$q_2 \mapsto 1,$	$q_3 \mapsto \langle 0,1,1 \rangle,$	$q_4 \mapsto \langle 0,1,0 \rangle$	1
b	$q_1, (q_3, 1), (q_4, 2), (q_3, 3), (q_4, 3)$	$q_1 \mapsto 1,$	$q_2 \mapsto 0,$	$q_3 \mapsto \langle 1,0,1 \rangle,$	$q_4 \mapsto \langle 0,1,1 \rangle$	1
a	$q_1, q_2, (q_4, 1), (q_3, 3), (q_4, 3)$	$q_1 \mapsto 1,$	$q_2 \mapsto 1,$	$q_3 \mapsto \langle 0,0,1 \rangle,$	$q_4 \mapsto \langle 1,0,1 \rangle$	1
a	$q_1, q_2, (q_3, 1), (q_3, 2), (q_4, 3)$	$q_1 \mapsto 1,$	$q_2 \mapsto 1,$	$q_3 \mapsto \langle 1,1,0 \rangle,$	$q_4 \mapsto \langle 0,0,1 \rangle$	1
a	$q_1, q_2, (q_3, 1), (q_4, 1), (q_4, 2)$	$q_1 \mapsto 1,$	$q_2 \mapsto 1,$	$q_3 \mapsto \langle 1,0,0 \rangle,$	$q_4 \mapsto \langle 1,1,0 \rangle$	1
a	$q_1, q_2, (q_3, 1), (q_4, 1), (q_3, 2), (q_3, 3)$	$q_1 \mapsto 1,$	$q_2 \mapsto 1,$	$q_3 \mapsto \langle 1,1,1 \rangle,$	$q_4 \mapsto \langle 1,0,0 \rangle$	0
b	$q_1, (q_3, 1), (q_4, 1), (q_3, 2), (q_4, 2), (q_4, 3)$	$q_1 \mapsto 1,$	$q_2 \mapsto 0,$	$q_3 \mapsto \langle 1,1,0 \rangle,$	$q_4 \mapsto \langle 1,1,1 \rangle$	1
b	$q_1, (q_4, 1), (q_3, 2), (q_4, 2), (q_3, 3)$	$q_1 \mapsto 1,$	$q_2 \mapsto 0,$	$q_3 \mapsto \langle 0,1,1 \rangle,$	$q_4 \mapsto \langle 1,1,0 \rangle$	1
b	$q_1, (q_3, 2), (q_4, 2), (q_3, 3), (q_4, 3)$	$q_1 \mapsto 1,$	$q_2 \mapsto 0,$	$q_3 \mapsto \langle 0,1,1 \rangle,$	$q_4 \mapsto \langle 0,1,1 \rangle$	1
b	$q_1, (q_4, 2), (q_3, 3), (q_4, 3)$	$q_1 \mapsto 1,$	$q_2 \mapsto 0,$	$q_3 \mapsto \langle 0,0,1 \rangle,$	$q_4 \mapsto \langle 0,1,1 \rangle$	1
b	$q_1, (q_3, 3), (q_4, 3)$	$q_1 \mapsto 1,$	$q_2 \mapsto 0,$	$q_3 \mapsto \langle 0,0,1 \rangle,$	$q_4 \mapsto \langle 0,0,1 \rangle$	1
b	$q_1, (q_4, 3)$	$q_1 \mapsto 1,$	$q_2 \mapsto 0,$	$q_3 \mapsto \langle 0,0,0 \rangle,$	$q_4 \mapsto \langle 0,0,1 \rangle$	1
b	q_1	$q_1 \mapsto 1,$	$q_2 \mapsto 0,$	$q_3 \mapsto \langle 0,0,0 \rangle,$	$q_4 \mapsto \langle 0,0,0 \rangle$	0

Fig. 3. Execution of the NCA and the NBVA for the regular expression `.*a(..){2,3}`.

Example 3.3. We consider the regex $r = \Sigma^* a(\Sigma\Sigma)\{2,3\}$. This is the same as `.*a(..){2,3}` in POSIX notation. This regex is recognized by the automata (NCA and NBVA respectively) of Fig. 3. The table also shows an example execution of the automata.

NBVAs are conceptually nondeterministic, as they allow several transitions to be enabled from each state. However, the presented semantics using configurations is ***deterministic*** and hence directly implementable. This can be seen clearly in Example 3.2 (Fig. 2) and Example 3.3 (Fig. 3). So, we do not need to determinize an NBVA in order to perform matching.

Example 3.4. The regex $r = \Sigma^* \sigma_1(\sigma_2\sigma_3\sigma_4)\{m, n\}\sigma_5$ with $1 \leq m \leq n$ is recognized by the following automata (NCA and NBVA):



We use the notation $v[m..n] \neq \mathbf{0}$ as abbreviation for the test $v[m] = 1 \vee \cdots \vee v[n] = 1$ (which is encoded as an action of type $\mathbb{B}^n \to \mathbb{B}$). For a pure state q that is indicated as being final with a double circle, $F(q) : \mathbb{B} \to \mathbb{B}$ is typically omitted and it is assumed to be the identity function.

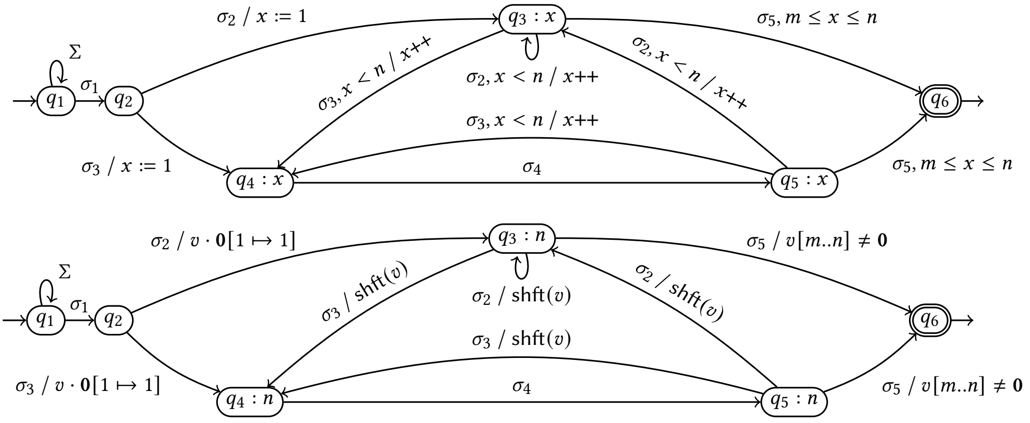

Σ
σ_1
q_1
q_2
$\sigma_2 \;/\; x := 1$
$q_3 : x$
$\sigma_5, m \leq x \leq n$
$\sigma_3, x < n \;/\; x{+}{+}$
$\sigma_2, x < n \;/\; x{+}{+}$
$\sigma_2, x < n \;/\; x{+}{+}$
q_6
$\sigma_3, x < n \;/\; x{+}{+}$
$\sigma_3 \;/\; x := 1$
$q_4 : x$
σ_4
$q_5 : x$
$\sigma_5, m \leq x \leq n$
Σ
σ_1
q_1
q_2
$\sigma_2 \;/\; v \cdot \mathbf{0}[1 \mapsto 1]$
$q_3 : n$
$\sigma_5 \;/\; v[m..n] \neq \mathbf{0}$
$\sigma_3 \;/\; \mathsf{shft}(v)$
$\sigma_2 \;/\; \mathsf{shft}(v)$
$\sigma_2 \;/\; \mathsf{shft}(v)$
q_6
$\sigma_3 \;/\; \mathsf{shft}(v)$
$\sigma_3 \;/\; v \cdot \mathbf{0}[1 \mapsto 1]$
$q_4 : n$
σ_4
$q_5 : n$
$\sigma_5 \;/\; v[m..n] \neq \mathbf{0}$

Fig. 4. NCA and NBVA for the regular expression $r = \Sigma^*\sigma_1(\sigma_2 \,|\, \sigma_3\sigma_4)\{m,n\}\sigma_5$ with $1 \leq m \leq n$.

Example 3.5. The regex $r = \Sigma^*\sigma_1(\sigma_2 \,|\, \sigma_3\sigma_4)\{m,n\}\sigma_5$ with $1 \leq m \leq n$ is recognized by the automata (NCA and NBVA) shown in Fig. 4. An important difference from previous examples is that the states q_3 and q_4 have 3 incoming arrows each. As we will discuss later, this corresponds to a merge operation on bit vectors that can be computationally challenging.

The operations on bit vectors in NBVAs (setting the lowest-order bit, shifting, reading high-order bits) is similar to the use of buffers and *sliding windows* in runtime verification with metric temporal logic (MTL) [Chattopadhyay and Mamouras 2020; Mamouras et al. 2021a,b; Mamouras and Wang 2020]. MTL has temporal connectives that specify time durations using intervals of the form $[m,n]$. These connectives are similar to the bounded repetition operators $\{m,n\}$ used in regexes. This similarity suggests that NBVAs could be useful in the context of runtime verification.

Translation from NCA to NBVA. We consider an NCA $\mathcal{A} = (Q, R, \Delta, I, F)$ whose states have at most one counter register. That is, $R(q)$ is either empty or a singleton set for every control state $q \in Q$. We will define the NBVA $\mathcal{A}' = (Q, w, \Delta', I', F')$ to have the same set of control states. If $R(q) = \emptyset$, then we put $w(q) = 1$ (pure state). If $R(q)$ is a singleton set, then we put $w(q) = n$, where n is the bound for the counter values of q (recall that we only consider NCAs with bounded counters). Suppose that $(p, \sigma, \varphi, q, \vartheta)$ is a transition of $\mathcal{A}$. When p is pure we can assume w.l.o.g. that φ is true (i.e., φ is equal to $R(p) \to \mathbb{N}$), otherwise the transition could be removed from the NCA. We will examine four cases, based on whether p and q are pure or not.

- Case: both p and q are pure. Both the guard φ and the action ϑ are trivial. We put the transition $(p, \sigma, q, \vartheta')$ in $\mathcal{A}'$, where $\vartheta'(0) = 0$ and $\vartheta'(1) = 1$ (i.e., ϑ' is the identity function).
- Case: only p is pure. The guard φ is trivial and the action ϑ specifies a constant counter value c. We put the transition $(p, \sigma, q, \vartheta')$ in $\mathcal{A}'$, where $\vartheta'(0) = \mathbf{0}$ and $\vartheta'(1) = \mathbf{0}[c \mapsto 1]$, the vector which is zero everywhere except for position c.
- Case: only q is pure. The action ϑ is trivial. We put the transition $(p, \sigma, q, \vartheta')$ in $\mathcal{A}'$, where

$$\vartheta'(v) = \begin{cases} 1, & \text{if } v[i] = 1 \text{ for some } i \in \varphi \\ 0, & \text{otherwise.} \end{cases}$$

- Case: none of p, q is pure. We put the transition $(p, \sigma, q, \vartheta')$ in $\mathcal{A}'$, where

$$\vartheta'(v) = \sum \Big\{ \mathbf{0}[\vartheta(i) \mapsto 1] \mid 1 \leq i \leq w(p) \text{ such that } i \in \varphi \text{ and } v[i] = 1 \Big\}.$$

Recall that the symbol $\sum$ is bitwise OR (disjunction).

We consider now the definition of the initialization function I'. For a pure state q, we define $I'(q) = 1$ if q is initial and $I'(q) = 0$ if q is not initial. For a counting state q, we define

$$I'(q)[i] = \begin{cases} 1, & \text{if } i \in I(q) \\ 0, & \text{otherwise} \end{cases}$$

for every position i with $1 \leq i \leq w(q)$.

Finally, we will define the finalization function F'. For a pure state q, we define the function $F'(q) : \mathbb{B} \longrightarrow \mathbb{B}$ as follows:

$$F'(q)(v) = \begin{cases} 1, & \text{if } v = 1 \text{ and } q \text{ is final} \\ 0, & \text{otherwise.} \end{cases}$$

For a counting state q, we set $F'(q)(v) = 1$ iff there is some i with $v[i] = 1$ such that $i \in F(q)$.

Lemma 3.6. *Let $\mathcal{A} = (Q, R, \Delta, I, F)$ be an NCA that uses at most one counter register. Then, the NBVA that results from translating $\mathcal{A}$ (according to the previous paragraphs) is equivalent to $\mathcal{A}$, i.e., it recognizes the same language as $\mathcal{A}$.*

Proof. Let $\mathcal{B} = (Q, w, \Delta', I', F')$ be the NBVA that results from the translation. The equivalence is seen by using the configuration semantics for NCAs. For a state q, the bit vector v for q records the set of active tokens on the state. More specifically, the following hold for the execution of $\mathcal{A}$ and $\mathcal{B}$ on the same input string $x \in \Sigma^*$:

(1) If q is a pure state, then $q \in [\![\mathcal{A}]\!](x)$ iff $[\![\mathcal{B}]\!](x)(q) = 1$.
(2) If q is a counting state and i is a position i with $1 \leq i \leq w(q)$, then

$$(q, i) \in [\![\mathcal{A}]\!](x) \iff [\![\mathcal{B}]\!](x)(q)[i] = 1.$$

The proof proceeds by induction on the input string x. For the base case $x = \varepsilon$, we recall that the initial configuration of $\mathcal{A}$ is equal to $[\![\mathcal{A}]\!](\varepsilon)$ and it consists of all initial tokens.

(1) Let q be a pure state. Then, $q \in [\![\mathcal{A}]\!](\varepsilon)$ iff q is initial iff $I'(q) = 1$ iff $[\![\mathcal{B}]\!](\varepsilon)(q) = 1$.
(2) Let q be a counting state and i be a position with $1 \leq i \leq w(q)$. Then, $(q, i) \in [\![\mathcal{A}]\!](\varepsilon)$ iff (q, i) is an initial token iff $i \in I(q)$ iff $I'(q)[i] = 1$ iff $[\![\mathcal{B}]\!](\varepsilon)(q)[i] = 1$.

For the step case, we consider the input string xa. There are many cases to examine for the transitions and the details are messy and not particularly interesting. For this reason, we will show the argument for the case of a counting state q, for which there are incoming transitions only from counting states. For an arbitrary position j with $1 \leq j \leq w(q)$, the following are equivalent:

(i) $(q, j) \in [\![\mathcal{A}]\!](xa)$
(ii) $(q, j) \in \delta([\![\mathcal{A}]\!](x), a)$
(iii) $(p, i) \to^a (q, j)$ for some token $(p, i) \in [\![\mathcal{A}]\!](x)$
(iv) there exist a transition $(p, \sigma, \varphi, q, \vartheta) \in \Delta$ and i s.t. $a \in \sigma$, $i \in \varphi$, $j = \vartheta(i)$ and $(p, i) \in [\![\mathcal{A}]\!](x)$ [*Note*: from the I.H., we know that $(p, i) \in [\![\mathcal{A}]\!](x)$ iff $[\![\mathcal{B}]\!](x)(p)[i] = 1$]
(v) there exists a transition $(p, \sigma, q, \vartheta') \in \Delta'$ s.t. $a \in \sigma$ and $\vartheta'([\![\mathcal{B}]\!](x)(p))[j] = 1$
(vi) $[\![\mathcal{B}]\!](xa)(q)[j] = 1$

This special case shows the main idea of the argument, so we leave the other cases to the reader. Now that we have proved the main claim, we can prove the lemma. Let $x \in \Sigma^*$ be an arbitrary string. The string x is accepted by $\mathcal{A}$ iff $[\![\mathcal{A}]\!](x)$ contains some final token (q, i) (i.e., $i \in F(q)$) iff $[\![\mathcal{B}]\!](x)(q)[i] = 1$ for some token (q, i) with $i \in F(q)$ iff $F'(q)([\![\mathcal{B}]\!](x)(q)) = 1$ for some state q iff $\mathcal{B}$ accepts x. □

Relationship between NCAs and NBVAs. The NCAs (with bounded counters) that we defined in Section 2 and the NBVAs of this section recognize exactly the regular languages, and therefore have the same expressiveness. Both models are extensions of NFAs. So, they recognize at least the regular languages. Moreover, both models are finite-state, because we consider *bounded* NCAs and NBVAs with *bounded-length* bit vectors, and therefore recognize exactly the class of regular languages.

Another relevant question is whether we can translate from one model to the other without a substantial blowup in size. Informally, Definition 3.1 (NBVAs) allows the transition to depend, not only on the counter valuations independently, but on the combination of active counter valuations. In order to make the correspondence between NCAs and NBVAs more direct, we can require that the actions ϑ in NBVAs are *linear*, that is: $\vartheta(\mathbf{0}) = \mathbf{0}$ and $\vartheta(u + v) = \vartheta(u) + \vartheta(v)$ for all bit vectors u, v (where + is bitwise OR). Moreover, we can require that $F(q) : \mathbb{B}^{w(q)} \longrightarrow \mathbb{B}$ is linear for every state q. We say that F is linear if $F(q)$ is linear for every state q. The linearity of the finalization function implies that $F(q)$ is determined uniquely by the values $F(q)(\mathbf{0}[i \mapsto 1]) = f_{q,i}$ for each $i \in \{1, \ldots, w(q)\}$. Using linearity, we see that

$$F(q)(v) = F(q)\big(\bigvee_{v[i]=1} \mathbf{0}[i \mapsto 1]\big) = \bigvee_{v[i]=1} f_{q,i} = \bigvee_{i=1}^{w(q)} \big(v[i] \wedge f_{q,i}\big).$$

So, the function $F(q)$ is determined by the bit vector $[f_{q,1}, f_{q,2}, \ldots, f_{q,w(q)}]$, which is of length $w(q)$. All the NBVA actions and finalization functions that we consider in the examples of Section 3 are linear. We also observe that Definition 2.1 (NCAs) restricts each transition $(p, \sigma, \varphi, q, \vartheta)$ to a deterministic action ϑ on counter valuations. That is, if the transition is enabled, then the current counter valuation is transformed into the next counter valuation. If the action is allowed to be nondeterministic, then the correspondence to NBVAs becomes more direct.

4 ALGORITHMS FOR EFFICIENT REGEX MATCHING

For regular expressions without bounded repetition (i.e., classical regular expressions with concatenation, choice and iteration), Thompson's algorithm provides a strong efficiency guarantee: the running time at each step, i.e., to process a single input symbol, is $O(m)$, where m is the size of the regular expression. In general, there is no known algorithm for regexes with bounded repetition that can offer such a good worst-case time complexity. We propose here specialized algorithms for bounded repetition that offer efficiency similar to Thompson's algorithm for regexes of a specific (but widely occurring) form. First, we consider bounded repetitions of the form $(\sigma_0\sigma_1 \ldots \sigma_{k-1})\{m, n\}$, where each σ_i is a predicate (character class). We show in Theorem 4.1 that there is an algorithm for matching such regexes with time complexity $O(k)$ per step. Then, we continue to generalize this result in Theorem 4.3 to bounded repetitions of the form $r\{m, n\}$, where r is a counting-free regex that recognizes strings of some fixed length k, that is, $\mathcal{L}(r) \subseteq \Sigma^k$ for some $k \geq 0$. The time complexity in this more general case is $O(\ell)$ per step, where ℓ is the number of predicate (character class) occurrences in r. Notice that in both cases the time complexity is the same as the complexity that an NFA-based algorithm (e.g., Thompson's) would have on the counting-free regex r^*. So, our results imply that we deal with bounded repetition without any (asymptotic) performance penalty in terms of time per step compared to the classical (counting-free) setting.

One key idea for our algorithms is to use bit vectors as a compact data structure to manipulate the configurations of NCAs/NBVAs. For NCAs/NBVAs that result from regular expressions with bounded repetition, only the following operations arise:

(1) Creating a bit vector that is zero everywhere except for the lowest position. This results from the NCA action $x := 1$.

(2) Copying a bit vector from one control state to another. This operation corresponds to the identity NCA action $x := x$.
(3) Shifting a bit vector and filling in a zero at the lowest position. This corresponds to the back-edge of a counting loop, which involves the x++ NCA action.
(4) Merging bit vectors using the bitwise OR operation (disjunction). This arises whenever there are more than two incoming edges at a control state.

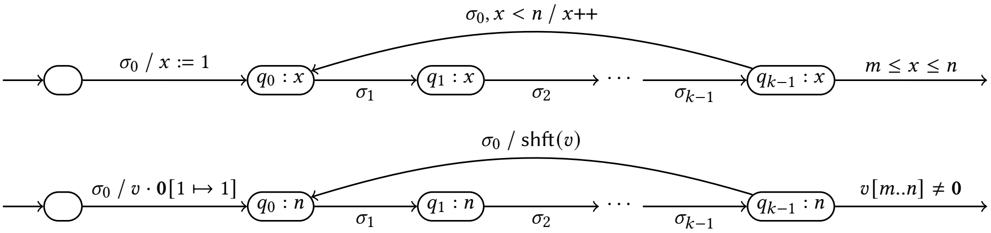
Example 3.5 shows an NBVA where the execution requires the merging of bit vectors that may contain a large number of 1's. These cases are difficult to handle when the size of the bit vectors is large, since the merging operation requires $O(n)$ time, where n is the bit vector size. Example 3.2, Example 3.3, and Example 3.4 show NBVAs where merging occurs, but it is easy to handle because one of the bit vectors has only one bit set. State q_3 of the NBVA in Example 3.2 merges the vector $\mathsf{shft}(v)$ with the vector $\mathbf{0}[1 \mapsto 1]$. This operation is the same as calculating $\mathsf{shft}(v)[1 \mapsto 1]$, which first shifts v and then sets the lowest position to 1. We say that this is a *trivial merge*, because it requires $O(1)$ time to perform, as we will see later in Section 4.1.

The algorithms that we will present in the rest of the section solve the problem of ***evaluating*** a regex r in the context of a bigger regex r', that is, when r appears as a sub-expression of r'. This is a slightly more general matching problem than top-level matching. Intuitively, at the top level, the matcher is initialized and started only once, at the beginning of the input stream. Within the context of a larger regex, the matcher for r can be "restarted" several times. This restarting corresponds to a token flowing into the corresponding sub-automaton. Consider, for example, the regex $(\Sigma \cdot \Sigma \cdot \Sigma)^* r$. The inner regex r is restarted every 3 characters. If we think of the corresponding automaton, a token flows into the sub-automaton for r every 3 characters. The functions that we will use for evaluation take two arguments: a Boolean value b and a symbol $a \in \Sigma$. The first argument $b \in \mathbb{B}$ indicates whether the matcher is restarted at the current step.

If a regex contains bounded repetitions that either have small repetition bounds (e.g., $r\{m, n\}$ with $m \leq n \leq 64$, where 64 is the size of a CPU register) or are of the forms considered in this section ("flat bounded repetition" and "loops of fixed length"), then the overall per-symbol time complexity is $O(\ell)$, where ℓ is the number of predicate (character class) occurrences in r. This upper bound matches the upper bound for classical regexes given by Thompson's algorithm.

4.1 Flat Bounded Repetition

Consider a bounded repetition of the form $r = (\sigma_0 \sigma_1 \ldots \sigma_{k-1})\{m, n\}$, which can occur as a sub-expression of a larger regular expression. The regular expression r corresponds to a sub-automaton (NCA and NBVA respectively) of the following form:



We observe that the only merging operation occurs at state q_0 and it is a trivial merge. For the NBVA, this merge corresponds to the computation of a bit vector $\mathsf{shft}(v) \vee \mathbf{0}[1 \mapsto 1]$, where $\vee$ is elementwise disjunction (i.e., bitwise OR). This can be implemented by first shifting v and then setting the bit at position 1 to 1. So, the efficiency of this computation solely depends on the efficiency of the shift operation. Another important issue is how to implement the test $v[m..n] \neq \mathbf{0}$

```
State:
   bit vectors $w_0, w_1, \dots, w_{k-1}$ of size $n$,
   initialized to $\mathbf{0}$
1  Fn Update1($b : \mathbb{B}, a : \Sigma$):
2      $w'_0 \leftarrow w_{k-1}$
3      for $i = 1, \dots, k-1$ do
4          $w'_i \leftarrow w_{i-1}$
5      $w'_0 \leftarrow \mathsf{shft}(w'_0)$
6      if $b = 1$ then
7          $w'_0[1] \leftarrow 1$
8      for $i = 0, \dots, k-1$ do
9          if $a \notin \sigma_i$ then
10             $w'_i \leftarrow \mathbf{0}$
11     for $i = 0, \dots, k-1$ do
12         $w_i \leftarrow w'_i$
13     return $\bigvee_{j=m}^{n} w_{k-1}[j]$
```

```
State:
   variable $s \in \{0, 1, \dots, k-1\}$,
   initialized to 0
   bit vectors $v_0, v_1, \dots, v_{k-1}$ of size $n$,
   initialized to $\mathbf{0}$
1  Fn Update2($b : \mathbb{B}, a : \Sigma$):
2      $s \leftarrow (s-1) \bmod k$
3      $v_s \leftarrow \mathsf{shft}(v_s)$
4      if $b = 1$ then
5          $v_s[1] \leftarrow 1$
6      for $i = 0, \dots, k-1$ do
7          if $a \notin \sigma_i$ then
8              $i' \leftarrow (s+i) \bmod k$
9              $v_{i'} \leftarrow \mathbf{0}$
10     $i \leftarrow (s-1) \bmod k$
11     return $\bigvee_{j=m}^{n} v_i[j]$
```

Fig. 5. Simple algorithms for matching subexpressions of the form $(\sigma_0\sigma_1 \dots \sigma_{k-1})\{m, n\}$.

efficiently, since the obvious implementation requires reading $n-m$ bits and therefore takes $\Theta(n-m)$ time. We will consider a sequence of algorithms that will lead to an efficient implementation.

Version 1. On the left-hand side of Fig. 5, we see a simple algorithm for implementing a bounded repetition of the form $(\sigma_0\sigma_1 \dots \sigma_{k-1})\{m, n\}$ that can be used in any context within a larger regular expression. This is a direct implementation of the semantics of the corresponding NBVA. The state of the algorithm consists of k bit vectors, each of size n, which we call $w_0, w_1, \dots, w_{k-1}$. At every step of the computation, the function Update1 is called. It takes as input a Boolean value b, which indicates the presence (value 1) or absence (value 0) of a transition from the rest of the automaton into the loop. The idea is that all the bit vectors have to rotate around the cycle by one position to the right (there is a back edge from q_{k-1} to q_0). If the current symbol $a \in \Sigma$ does not satisfy the predicate σ_i, then the bit vector w_i should become $\mathbf{0}$ at the end of the step. We use the bit vectors $w'_0, w'_1, \dots, w'_{k-1}$ as temporary variables, so as to not overwrite bit vectors that need to be used later. The case of the bit vector w_0 is the most interesting one, because control state q_0 is the "entry point" to the loop. In particular, it is (1) the target of the back edge of the loop, and (2) the target of transitions from the rest of the automaton into the loop. The combined effect of lines 2 & 5 (see pseudocode of Fig. 5) is that w'_0 is set to be equal to $\mathsf{shft}(w_{k-1})$, i.e., the result of shifting w_{k-1} (bit vector for q_{k-1} from the previous step) by one position. Moreover, if the Boolean variable b is equal to 1 ("true"), then $w'_0[1]$ is set to 1. The output of the function Update1 indicates whether there is a transition from the "exit" control state q_{k-1} to the rest of the automaton. This happens exactly when at least one of the bits $w_{k-1}[m], \dots, w_{k-1}[n]$ is equal to 1. This is why we specify the output value to be the finite disjunction $\bigvee_{j=m}^{n} w_{k-1}[j]$.

We see in Fig. 5 (left-hand side) that there are several operations that are performed on bit vectors in Update1. The assignments in lines 2, 4 and 12 are implemented as pointer assignments. The assignment $w'_0 \leftarrow \mathsf{shft}(w'_0)$ in line 5 is implemented as a shift operation on the bit vector data structure. The assignment $w'_0[1] \leftarrow 1$ in line 7 sets the lowest position of the bit vector to 1. The assignment $w'_i \leftarrow \mathbf{0}$ in line 10 sets the bit vector to $\mathbf{0}$. Finally, for the return expression of line 13, we need to support an operation that reads the bit from a specified position in a bit vector. Fig. 6

```
Fields of Data Structure:
    variable $start \in \{0, 1, \dots, n-1\}$
    variable $size \in \{0, 1, \dots, n\}$
    $n$-bit array $arr$ (0-based indexing)

Fn SetToZero():
    $start \leftarrow 0$
    $size \leftarrow 0$

Fn SetAtPos0($b : \mathbb{B}$):
    if $size = 0$ then
        $size \leftarrow 1$
    $arr[start] \leftarrow b$

Fn Shift():
    $start \leftarrow (start - 1) \bmod n$
    $arr[start] \leftarrow 0$
    if $size < n$ then
        $size \leftarrow size + 1$

Fn Get($i : \mathbb{N}$):
    if $i < size$ then
        $j \leftarrow (start + i) \bmod n$
        return $arr[j]$
    else
        return 0
```

Fig. 6. A data structure for n-bit vectors that provides efficient implementations for several operations.

shows the implementation of a data structure that can support all the needed operations. Each operation requires $O(1)$ time. Note that the data structure of Fig. 6 uses 0-based array indexing (for convenience of implementation), but in Fig. 5 we use 1-based indexing for the bit vectors. The main idea is to use a circular array to support efficient shifting. In order to set the array to zero without filling in 0's, we introduce the variable *size*. The idea is that the represented vector v has value 0 at all positions $i \geq size$. In particular, if $size = 0$, then the vector is equal to $\mathbf{0}$.

The function Update1 performs $O(k + (n - m))$ work at every step. The updating of each bit vector needs $O(1)$ time, because the assignments can be implemented with pointers (i.e., without copying the bit vectors) and the shift operation is implemented efficiently by representing a bit vector with a circular array. The generation of the output value takes $\Theta(n - m)$ time, because it requires reading $n - m$ bits. This cost, which is dependent on the repetition bounds, can be eliminated, as we will see later.

Version 2. On the right-hand side of Fig. 5, we see a small modification of the algorithm for the bounded repetition $(\sigma_0\sigma_1 \dots \sigma_{k-1})\{m, n\}$. The bit vectors $v_0, v_1, \dots, v_{k-1}$ are not copied, but instead a variable $s \in \{0, \dots, k-1\}$ gives the index that corresponds to the "entry" control state q_0. In general, the index $(s + i) \bmod k$ corresponds to the control state q_i. Notice that

$$(s + (k-1)) \bmod k = (s - 1) \bmod k$$

is the index that corresponds to the "exit" control state q_{k-1}. More precisely, this algorithm maintains the following correspondence with the previous one:

$$w_i = v_{i'}, \text{ where } i' = (s + i) \bmod k,$$

for every $i = 0, \dots, k-1$. The time complexity of Update2 is the same as that of Update1. It does avoid, however, the copying of bit vector pointers using this convenient trick with the indexes.

Version 3. We will see now in Fig. 7 how we can decrease the time complexity of the update function from $\Theta(k + (n - m))$, which depends on the repetition bounds m and n, to $\Theta(k)$, which is independent of the repetition bounds. The idea to accomplish this is to maintain an additional integer variable y_i for every bit vector v_i, which records the index of a witness for $\bigvee_{j=m}^{n} v_i[j]$ being equal to 1 ("true"). More specifically, we record the smallest index $j \geq m$ such that $v_i[j] = 1$. If no such index exists, then we put $y_i = \infty$. By recording this information, we can replace the computation of a disjunction $\bigvee_{j=m}^{n} v_i[j]$, which takes $\Theta(n - m)$ time, with the test $m \leq y_i \leq n$, which takes constant time. It remains to see that the values of these variables can be updated in

```
State:
  variable $s \in \{0, 1, \ldots, k-1\}$ (indicates the index for control state $q_0$), initialized to 0
  bit vectors $v_0, v_1, \ldots, v_{k-1}$ of size $n$ (one for each control state), initialized to $\mathbf{0}$
  integer variables $y_0, \ldots, y_{k-1} \in \{m, \ldots, n\} \cup \{\infty\}$, initialized to $\infty$
Fn Update3($b : \mathit{Bool}, a : \Sigma$):
    $s \leftarrow (s-1) \bmod k$   // rotate around the cycle by one position
    $v_s \leftarrow \mathrm{shft}(v_s)$   // shift the bit vector that is moving along the back edge
    if $b = 1$ then   // incoming transition to the “entry” control state
        $v_s[1] \leftarrow 1$
    if $v_s[m] = 1$ then
        $y_s \leftarrow m$   // new witness, which has value $m$
    else if $y_s = n$ then
        $y_s \leftarrow \infty$   // the (unique) witness falls off
    else   // no new witness, $y_s < n$ or $y_s = \infty$
        $y_s \leftarrow y_s + 1$   // increment the witness, note that $\infty + 1 = \infty$
    for $i = 0, \ldots, k-1$ do
        if $a \notin \sigma_i$ then   // transition to $q_i$ cannot be taken on symbol $a$
            $i' \leftarrow (s+i) \bmod k$   // index for control state $q_i$
            $v_{i'} \leftarrow \mathbf{0}$   // set bit vector to $\mathbf{0}$
            $y_{i'} \leftarrow \infty$   // no witness
    $i \leftarrow (s-1) \bmod k$   // index for control state $q_{k-1}$
    return $m \leq y_i \leq n$
```

Fig. 7. Efficient algorithm for matching subexpressions of the form $(\sigma_0\sigma_1 \ldots \sigma_{k-1})\{m, n\}$.

constant time per step. The main observation is that, at each step, only one bit vector changes, namely the one that moves from the “exit” control state q_{k-1} to the “entry” control state q_0 via the back edge. This bit vector is shifted and potentially one bit is set to 1 (if the argument b is 1). The variable y_s is updated in time $O(1)$ to reflect these changes. The output is computed in constant time with the Boolean expression $(m \leq y_i \leq n)$, where $i = (s-1) \bmod k$ is the index corresponding to the control state q_{k-1}. It follows that the total running time for every invocation of Update3 (Fig. 7) is $O(k)$, which is independent of the repetition bounds.

Theorem 4.1 (**Efficient Flat Loops**). *Let r be a regular expression of the form*

$$(\sigma_0\sigma_1 \ldots \sigma_{k-1})\{m, n\},$$

where each σ_i is a predicate over the alphabet Σ. There is an algorithm for evaluating r (where r can be used in any context within a larger regular expression) with time complexity $O(k)$ for every step.

Proof. As we have already discussed, the algorithm of Fig. 7, which is based on NBVA execution, achieves the time complexity bound of $O(k)$ per step. This relies on an efficient implementation of the shift operation on bit vectors. We have already mentioned that this can be achieved by using circular arrays for representing bit vectors, where shifting corresponds to decrementing (modulo n, the size of the bit vector) the index for the first element. □

Memory Optimizations. Each bit vector of the algorithm of Fig. 7 can be implemented with an array of n bits, where n is its size. An additional $\Theta(\log_2 n)$ bits are needed to store an index within the array, in order to efficiently implement the shift operation. This extra storage for the index is negligible, as it corresponds to one memory location for all practical instances. This implementation makes effective use of memory for *dense* bit vectors, that is, bit vectors that have a large number of digits equal to 1.

For the case of sparse bit vectors, one may want to consider an implementation that uses a list of indexes for the 1's. If there are K 1's, then this representation needs at least $K \cdot \log_2 n$ bits. In fact, there may be overheads associated with lists that demand more memory, but we will ignore them for the sake of simplifying the analysis. If $K \cdot \log_2 n < n$ or, equivalently, $K < n/\log_2 n$, then it may make sense to consider a "sparse" representation. This is the approach taken in [Turoňová et al. 2020]. We emphasize that choosing a sparse representation does not improve the time complexity. It can only reduce the memory footprint when K is smaller than $n/\log_2 n$. If $K \geq n/\log_2 n$, then the "dense" representation with bit vectors is superior. In any case, the algorithms of Fig. 5 and Fig. 7 can be easily made to work with either representation of bit vectors.

4.2 Loops of fixed length

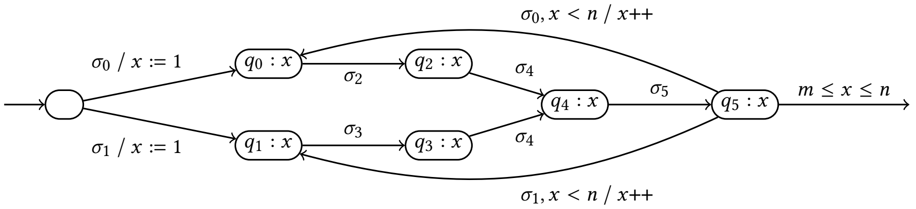
We say that a regular expression r is of *fixed length* if there exists a nonnegative integer $k \geq 0$ such that $\mathcal{L}(r) \subseteq \Sigma^k$. That is, all strings recognized by r are of the same length. A bounded repetition $r\{m, n\}$ is said to be a *fixed-length loop* if r is of fixed length. Some examples of fixed-length loops are seen below:

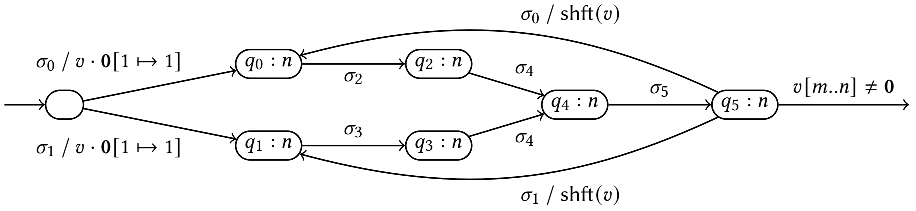
$$\sigma\{m,n\} \qquad (\sigma_0 \,|\, \sigma_1)\{m,n\} \qquad (\sigma_0\sigma_1)\{m,n\} \qquad (\sigma_0\sigma_2 \,|\, \sigma_1\sigma_3)\{m,n\}$$

$$(\sigma_0\sigma_1\sigma_2)\{m,n\} \quad (\sigma_0\sigma_2\sigma_4 \,|\, \sigma_1\sigma_3\sigma_5)\{m,n\} \quad (\sigma_0(\sigma_1\sigma_3 \,|\, \sigma_2\sigma_4))\{m,n\} \quad ((\sigma_0\sigma_2 \,|\, \sigma_1\sigma_3)\sigma_4)\{m,n\}$$

We will see that the algorithm of Fig. 7 can be generalized to work with fixed-length loops. This is a strict generalization, as the examples above indicate.

Example 4.2. Consider the regular expression $((\sigma_0\sigma_2 \,|\, \sigma_1\sigma_3)\sigma_4\sigma_5)\{m, n\}$, which is a fixed-length loop. We show below the sub-automaton (NCA and NBVA respectively) that this regular expression would correspond to if it were used in the context of a larger regular expression:




$\sigma_0 \;/\; \mathsf{shft}(v)$
$\sigma_0 \;/\; v \cdot \mathbf{0}[1 \mapsto 1]$
$q_0 : n$
σ_2
$q_2 : n$
σ_4
$q_4 : n$
σ_5
$q_5 : n$
$v[m..n] \neq \mathbf{0}$
$\sigma_1 \;/\; v \cdot \mathbf{0}[1 \mapsto 1]$
$q_1 : n$
σ_3
$q_3 : n$
σ_4
$\sigma_1 \;/\; \mathsf{shft}(v)$

We have numbered the states in "columns" from left to right. Column 0 consists of q_0 and q_1, the states that are reached in 1 step. Column 1 consists of q_2 and q_3, the states that are reached in 2 steps. Column 2 (resp., column 3) consists of q_4 (resp., q_5), the state that is reached in 3 (resp., 4) steps.

THEOREM 4.3 (**EFFICIENT FIXED-LENGTH LOOPS**). *Let r be a counting-free regular expression of the form $r\{m, n\}$, where r is a fixed-length regular expression. There is an algorithm for evaluating r (where r can be used in any context within a larger regular expression) with time complexity $O(\ell)$ for every step, where ℓ is the number of predicate occurrences in r.*

```
State:
    (there are $k$ columns and $\ell$ control states, which means that $\ell \geq k$)
    variable $s \in \{0, 1, \ldots, k-1\}$ (indicates the index for column 0), initialized to 0
    bit vectors $w_0, w_1, \ldots, w_{k-1}$ of size $n$ (one for each column), initialized to $\mathbf{0}$
    bit vectors $v_0, v_1, \ldots, v_{\ell-1} \in \{0, 1\}$, initialized to $\mathbf{0}$
    integer variables $y_0, y_1, \ldots, y_{\ell-1} \in \{m, \ldots, n\} \cup \{\infty\}$ for "witnesses", initialized to $\infty$
1 Fn Update($b : Bool, a : \Sigma$):
2     Update the bit vectors $v_0, \ldots, v_{\ell-1}$ and the variables $y_0, y_1, \ldots, y_{\ell-1}$ as if they were tokens
      of automata, using $b$ and $a$ (merge the bit vectors with bitwise OR & the integer witnesses
      with min). (We discuss in Theorem 4.3 how this can be done efficiently.)
3     $s \leftarrow (s-1) \bmod k$   // rotate columns around the cycle by one position
4     $w_s \leftarrow \mathsf{shft}(w_s)$   // shift the bit vector that is moving along the back edge
5     if $b = 1$ then   // incoming transition to the "entry" control state
6         $w_s[1] \leftarrow 1$
7     Let $J \subseteq \{0, \ldots, \ell-1\}$ be the set of state indexes for "exit" control states.
8     return $\bigvee_{j \in J} (m \leq y_j \leq n)$
```

Fig. 8. Efficient algorithm for matching subexpressions of the form $r\{m, n\}$, where r is of fixed length k.

Proof. We invite the reader to use the automata of Example 4.2 in order to understand the main ideas in this proof. Let s_i be the *size* variable (of the data structure described in Fig. 6) for the bit vector v_i. The key observation is that we only need to store one bit array per column of states (recall from Example 4.2 what state columns are). This is because for every control state q_i of some column j the following holds: the bit vector v_i is determined by the size variable s_i and the bit vector w_j for column j. More precisely, the following property holds: $v_i[t] = w_j[t]$ for every $t = 1, \ldots, s_i$ and $v_i[t] = 0$ for $t > s_i$. When this property holds, we say that v_i is column-compatible. Notice that column-compatibility means that we do not need to store a bit array for v_i, since v_i is determined by s_i and w_j. The column-compatibility property can be proved by induction. In the beginning, all states are inactive and their bit vectors are equal to $\mathbf{0}$. When a symbol is consumed, each bit vector goes from some state of column j to some state of column $(j+1) \bmod k$. If any merging of bit vectors happens, then the property is preserved, because the merging of column-compatible bit vectors produces a column-compatible bit vector. For column 0, every state merges the bit vectors from column $k-1$ (by the I.H., they are column-compatible) and sets the value at position 1 to 1 (if a transition from the environment to the entry states occurs). So, the property of column-compatibility is preserved for column 0 as well. The column-compatibility property justifies a key optimization: removing the bit arrays from the (per state) vectors $v_0, v_1, \ldots, v_{\ell-1}$ and using the corresponding bit arrays from the (per column) vectors $w_0, w_1, \ldots, w_{k-1}$. This optimization means that the bit vector merging (see line 2 in Fig. 8) can involve only the *size* variables from the data structures (taking their maximum) without performing any operations on bit arrays. This makes the merging a constant-time operation. It follows that this optimized algorithm performs $O(\ell)$ work (for line 2) at each step. So, the total running time per step is $O(\ell)$. □

5 EXPERIMENTS

In this section, we compare the performance of our regex engine, BVA-Scan, against three open-source engines: RE2 [RE2 2023], PCRE [Hazel and Herczeg 2022], and CA [Turoňová et al. 2020]. RE2 is a state-of-the-art regex engine that is based on DFAs and NFAs. The main idea for the practical efficiency of RE2 is that it generates and caches the DFA for the regular expression on the fly. If the generated DFA stays within a predetermined size, then each step involves one memory lookup and is very efficient. If the DFA becomes too large, then the engine may end up reverting to an NFA simulation algorithm, which is substantially slower. PCRE is a standard backtracking engine. This means that some patterns and input strings can give rise to exponential running time (in the length of the input string). The average behavior of PCRE, however, is much better than what the worst case suggests. CA is an engine based on the simulation of the execution of a variant of deterministic counter automata ("counting-set automata"). CA is able to handle a subclass of regexes with large repetition bounds in constant time. However, for some regexes, CA's determinization algorithm overapproximates the language defined by the regular expression.

We have evaluated the performance of BVA-Scan, RE2, PCRE and CA with both microbenchmarks (Section 5.1) and benchmarks that contain regexes used in real-world applications (Section 5.2).

BVA-Scan. We have implemented BVA-Scan using the Rust programming language. Our implementation is based on the deterministic execution of NBVAs using bit vectors (see Section 3), and it incorporates the efficient algorithms for bounded repetition from Section 4. For a bounded repetition $r\{m, n\}$, where r is not flat (Theorem 4.1) or of fixed length (Theorem 4.3), it is still possible to provide an efficient implementation using NBVAs when n is not too large. For example, if $n \leq 64$ then each bit vector can fit in a CPU register and therefore all NBVA operations (shift, merge, set, read) can be performed in $O(1)$ time. This approach is much better than naive unfolding, which is the same as expanding an NCA into an NFA, as it takes advantage of the bit parallelism that the CPU can offer. We have also included other performance optimizations that are commonly found in regex engines, such as an efficient algorithm for string search. These optimizations are helpful because many regexes that arise in real applications are of a relatively simple form.

Experimental setup. The experiments were executed in Ubuntu 20.04 on a desktop computer equipped with an Intel(R) Xeon(R) W-2295 CPU (18 cores) and 128 GB of RAM. We used Rust 1.59.0, GCC/G++ 9.4.0, and the CA binary downloaded from [Library 2021]. For each experiment, we executed 10 trials and we report the mean of the measurements.

5.1 Microbenchmarks

We have evaluated the performance of the algorithms presented in Section 4 using regexes of the following 8 forms: `a.{`k`}c`, `a(..){`k`}c`, `a.{0,`k`}c`, `a(..){0,`k`}c`, `a.{`k`,}c`, `a(..){`k`,}c`, `a.{`$k/2$`,`k`}c`, and `a(..){`$k/2$`,`k`}c`. We vary the values for the parameter k in order to examine the effect of the repetition bounds on the running time. These regexes are similar to patterns that are used in real-world applications. They are sometimes not supported by or cause timeouts in state-of-the-art engines like RE2 and PCRE, especially for larger values of k. We matched these regexes over an input string that contains 10 million characters. Each character is either the English letter 'a' or 'b', and is chosen uniformly at random. Therefore, the last predicate `c` in each microbenchmark regex ensures that all engines will be forced to search for a match over the entire input.

Fig. 9 shows the results for each benchmark. The black dashed line indicates the 10-second timeout limit that we have set. Each point in Fig. 9 is a measurement of the time it took for a regex engine to match a pattern over the input text. When an engine does not support a regex or the running time exceeds the 10-second timeout limit, then there is not a corresponding measurement. For example, RE2 does not support regexes with repetition bounds that exceed 1000. Therefore, when $k > 1000$, no point is shown for RE2. The CA engine does not support the regexes `a(..){`k`}c`, `a(..){0,`k`}c`, `a(..){`k`,}c`, and `a(..){`$k/2$`,`k`}c`, because it uses a model of counter automata that is more restricted than our NCA model of Section 2. Moreover, when k is small, we have observed that CA sometimes crashes and we are therefore unable to obtain a measurement (as seen in Fig. 9). The PCRE engine reaches the 10-second timeout limit on all microbenchmarks when k becomes large enough. Moreover, it times out on regexes `a.{`k`,}c` and `a(..){`k`,}c` for all values of k. These

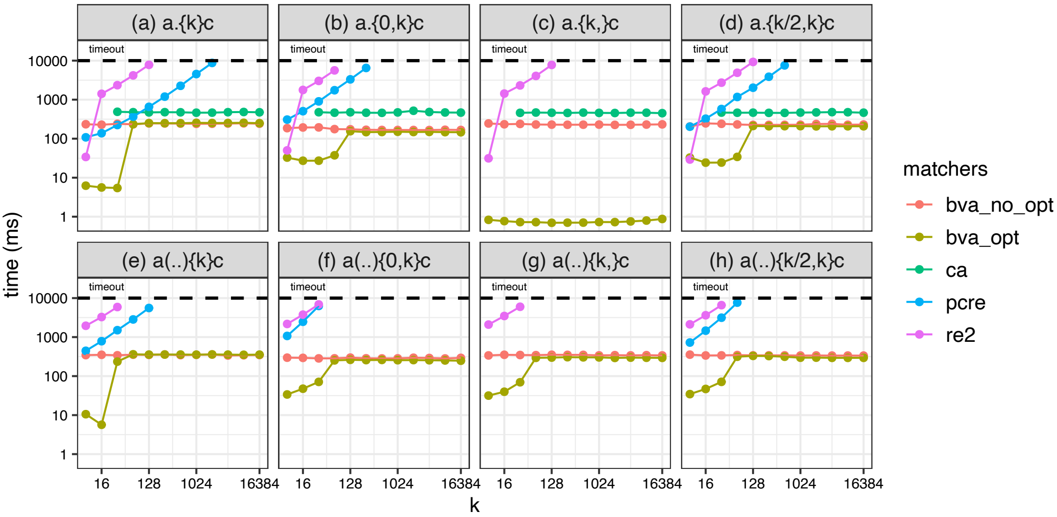


Fig. 9. Matching time (ms) of regexes in RE2, CA, PCRE, and BVA-Scan with and without optimizations for different values of k (timeout = 10 sec)

regexes are equivalent to a.{k}.*c and a(..){k}(..)*c respectively. They are similar to the pattern a.*c, which is known to cause catastrophic backtracking [Goyvaerts 2021].

The results of the microbenchmarks show that BVA-Scan (i.e., bva_opt) provides competitive performance against RE2 and PCRE when k is small (e.g., $k < 128$). For larger values of k, the speed of BVA-Scan remains constant as k increases. BVA-Scan is faster than CA by at least 2×, and faster than RE2 and PCRE by orders of magnitude. Notice that the performance of BVA-Scan (bva_opt) is better for smaller values of k (i.e., $k \leq 32$) compared to larger values of k (i.e., $k > 32$). This is because for smaller values of k the bit vectors used in our implementation fit entirely in CPU registers, which accelerates the bit vector operations that are performed during matching. Moreover, the matching speed of BVA-Scan is particularly fast for regex a.{k,}c (see Fig. 9(c)) as it provides an optimization for that regex pattern. More specifically, the matching of regex a.{k,}c, which is the same as a.{k}.*c, can be implemented by searching for the first occurrence of a, then skipping k letters, followed by a search for the letter c. We have also evaluated the performance of BVA-Scan with all optimizations disabled (i.e., bva_no_opt), in order to evaluate only the performance of the algorithms presented in Section 4.

5.2 Application Benchmarks

We have evaluated the performance of BVA-Scan using six benchmarks, which contain regexes collected from real applications. These benchmarks are: (1) the **Snort** [Roesch 1999; Snort 2023] and (2) **Suricata** benchmarks [Suricata 2023] that contain patterns for network traffic, (3) the **Protomata** benchmark that includes 1309 protein motifs from the PROSITE database [Roy and Aluru 2016; Sigrist et al. 2009], (4) the **ClamAV** benchmark [ClamAV 2023] that contains patterns that indicate the presence of viruses, (5) the **SpamAssassin** benchmark [SpamAssassin 2022] that includes patterns for detecting spam email, and (6) the **RegexLib** benchmark [RegexLib 2023] that is a collection of regexes for describing email addresses, phone numbers, and so on. In total, we have collected around 20,000 regexes.

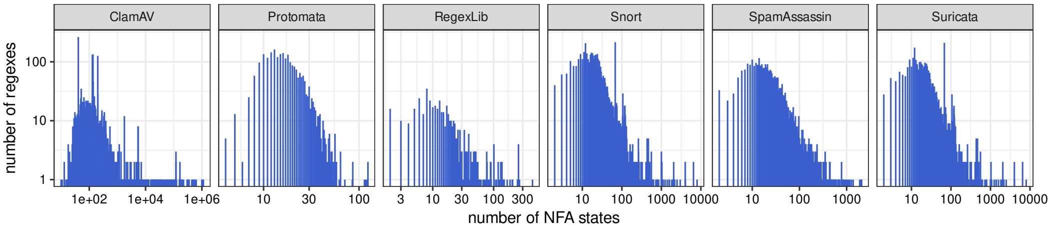


Fig. 10. Distribution of the number of NFA states generated by unfolding regexes

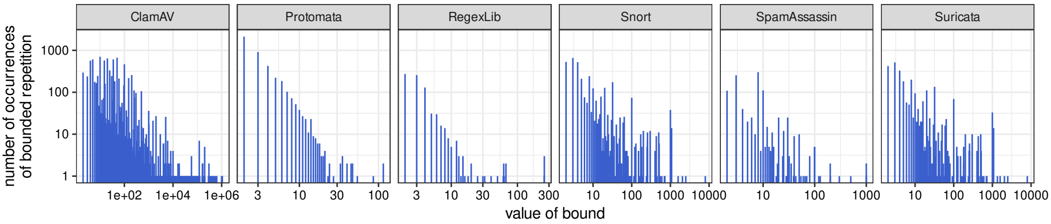


Fig. 11. Distribution of upper repetition bounds in regexes

5.2.1 Analysis of Regex Datasets. In the Snort, Suricata, SpamAssassin, and RegexLib benchmarks, some of the collected regexes may contain backreferences [Backreferences 2022], which are not regular constructs (i.e., they can give rise to non-regular languages). We filter out regexes with backreferences from the source datasets (about 2% of all regexes) and we eliminate duplicates. The benchmarking datasets contain the remaining regexes (64% of them contain bounded repetition).

Fig. 10 shows the distribution of the number of NFA states needed for the regexes of each dataset. For each regex, the number of NFA states is calculated by unfolding the bounded repetitions, e.g., $(ab)\{3\}$ is unfolded into $ababab$. The horizontal axis is the number of NFA states, and the vertical axis is the count of regexes. We observe that there are many regexes whose NFAs have a large number of states. For such regexes, it would be computationally expensive to perform matching using NFA simulation.

Fig. 11 shows the distribution of the upper repetition bounds of bounded repetitions. For the patterns $r\{n\}$ (which is abbreviation for $r\{n, n\}$), $r\{m, n\}$ and $r\{n, \}$ (which is equivalent to $r\{n\}r^*$), the value n is the upper repetition bound. The horizontal axis of Fig. 11 is the upper repetition bound, and the vertical axis is the number of occurrences of bounded repetition. We observe that all the datasets have many regexes with large upper repetition bounds. BVA-Scan uses the algorithms described in Section 4 to deal effectively with such regexes.

5.2.2 Benchmarking Results. We measured the running time of matching for regexes in different application benchmarks. For each experiment we used an input sequence of 1 million bytes. We used two different methods to generate these input sequences: (1) by choosing each byte independently and uniformly at random and (2) using an adversarial input generator to construct the input. The adversarial input generator analyzes a regex in order to identify a set of characters that are likely to trigger a state transition during the execution of the automaton. Then, the generator randomly selects characters from this set to construct the input string. We measured the running time for regex matching using both of these methods of input generation.

Table 1 shows the ratios of unhandled cases of regex matching for RE2, CA, PCRE, and BVA-Scan, where unhandled means that (1) the regex can be parsed but is not supported for matching (e.g., the

Table 1. Ratios of unhandled cases of regex matching in different engines (x indicates the regex is unsupported, o means the matching time exceeds 10 seconds)

Benchmark	# RE2 x	# RE2 o	# CA x	# CA o	# PCRE x	# PCRE o	# BVA-Scan x	# BVA-Scan o
Snort	0.29%	0%	27.19%	0.21%	0%	1.60%	0%	0%
Suricata	0.35%	0%	27.99%	0.13%	0%	0.89%	0%	0%
Protomata	0%	0%	40.42%	28.57%	0%	0%	0%	0%
ClamAV	7.63%	0%	4.40%	6.37%	0%	1.18%	0%	0%
SpamAssassin	0%	0%	40.08%	1.31%	0%	1.28%	0%	0%
RegexLib	0%	0%	37.35%	1.81%	0%	1.41%	0%	0%

upper repetition bound is too large), or (2) the engine times out during matching (i.e., the running time exceeds 10 seconds).

RE2 does not support regexes with upper repetition bounds that exceed 1000. CA cannot handle a large number of regexes from the application benchmarks. We have observed that it fails (i.e., crashes with an exception) in many cases where the pattern is a string or it contains sub-patterns $r\{m, n\}$ with small repetition bounds, but the reason behind these failures is unclear. Moreover, CA does not support the matching of certain classes of patterns such as `.*(aa){`n`}` and `.*a{0,`m`}a{`n`}` due to the inherent limitations of the automata model and the determinization algorithm that it uses. CA provides consistent speed for regexes with large bounds in counting repetition. However, the matching of regexes that contain many occurrences of bounded repetition may cause a timeout. Several timeouts are reported for PCRE due to catastrophic backtracking. BVA-Scan always performs regex matching within 10 seconds, even for those regexes with huge repetition bounds.

Fig. 12 reports the running time (in milliseconds) for matching regexes (taken from 6 different application benchmarks) over a randomly generated input string. Each plot of Fig. 12 is a performance comparison between BVA-Scan and another engine. The plots are organized in a grid with $3 \times 6 = 18$ plots: each column of the grid corresponds to a different regex engine (CA, PRCRE, or RE2), and each row corresponds to a different application benchmark. Each plot in the column for engine E contains points for the regexes that can be handled by both BVA-Scan and E. For each point (t, t') the horizontal coordinate t (resp., the vertical coordinate t') is the matching time of BVA-Scan (resp., engine E). So, if a point is below the diagonal, then this means that BVA-Scan is faster than engine E. We calculate the percentage of regexes for which BVA-Scan is faster than E and present this information on each plot (see, e.g., "wins: 100.00%"). The results show that BVA-Scan is consistently faster than CA (100% cases), PCRE ($\geq$ 96.72%), and RE2 ($\geq$ 99.57%) for the vast majority of regexes in the benchmarks. Fig. 12 also reports the percentage of regexes for which there is no significant difference between BVA-Scan and the engine it is compared against. This information is relevant, because measurements of running time have some inherent uncertainty. The highlighted area in each plot (narrow band around the diagonal) contains those points for which the matching time of BVA-Scan is between 0.5× and 2× the matching time of the other engine. In general, the percentage of these regexes is below 5%.

Fig. 13 reports the matching time of regexes when the input text is produced with the adversarial generation method. The results show that BVA-Scan is faster than CA ($\geq$ 99.81%), PCRE ($\geq$ 99.35%), and RE2 ($\geq$ 99.56%), and the ratio of regexes that are close to the diagonal is below 2% in general.

Table 2 (resp., Table 3) reports the average speedup of BVA-Scan over the other engines for random (resp., adversarial) input. For matching over random input, the average speedup ranges from 125× to 627× for CA, 9× to 164× for PCRE, and 9× to 15× for RE2. For adversarial input, it ranges from 46× to 341× for CA, 41× to 125× for PCRE, and 11× to 36× for RE2.

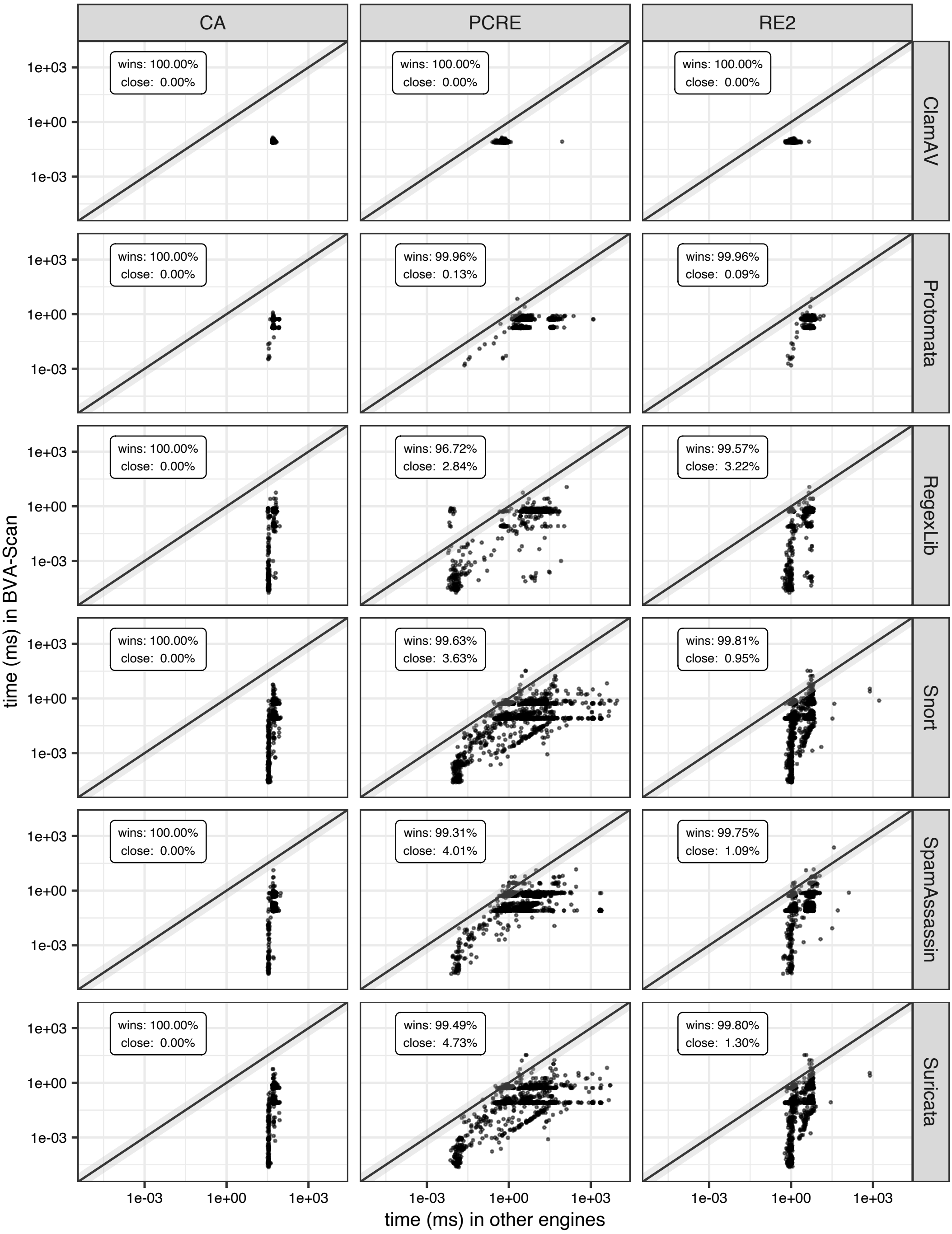


Fig. 12. Matching time (ms) of RE2, CA, PCRE and BVA-Scan over regexes collected from real-world applications for random input. The timeout is set to 10 seconds. The grey area contains points for which the matching time of BVA-Scan is between 0.5× and 2× the matching time of the other engine.

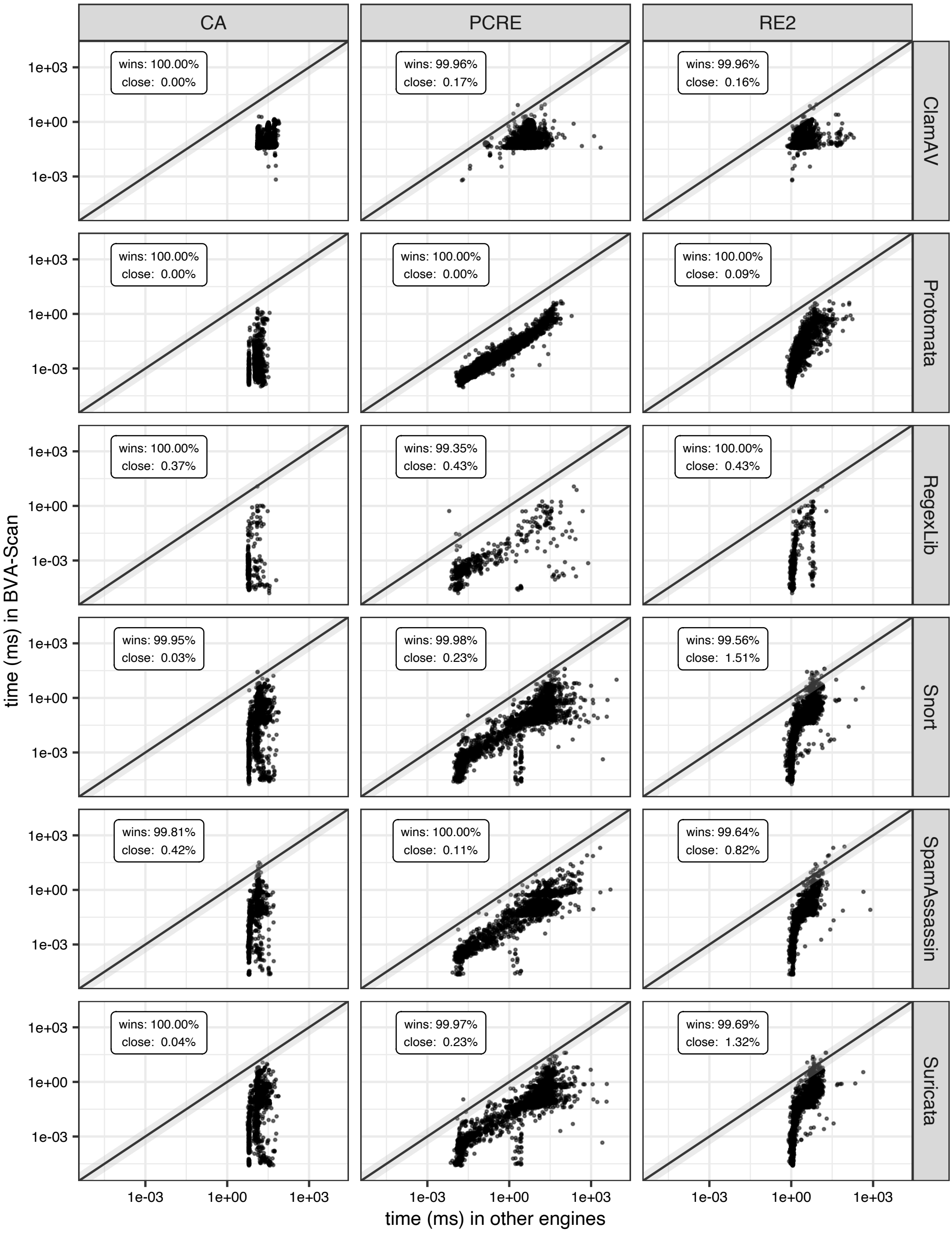


Fig. 13. Matching time (ms) of RE2, CA, PCRE and BVA-Scan over regexes collected from real-world applications for adversarial input. The timeout is set to 10 seconds. The grey area contains points for which the matching time of BVA-Scan is between 0.5× and 2× the matching time of the other engine.

Table 2. Average speedup of BVA-Scan versus CA, RE2, PCRE over different benchmarks for random input

	ClamAV	Protomata	RegexLib	Snort	SpamAssassin	Suricata
CA	627	125	142	400	217	394
PCRE	9	18	23	164	129	118
RE2	15	11	9	15	10	13

Table 3. Average speedup of BVA-Scan versus CA, RE2, PCRE over different benchmarks for adversarial input

	ClamAV	Protomata	RegexLib	Snort	SpamAssassin	Suricata
CA	160	341	109	62	46	68
PCRE	41	53	125	59	62	54
RE2	22	36	21	13	11	14

5.2.3 Benchmarking Difficult Regexes. We have compiled a list of challenging regexes from our application benchmarks in order to evaluate the benefit of the algorithms presented in Section 4. A challenging regex typically involves a large amount of *counting nondeterminism*. This means that, during matching over adversarial input, the configurations of the NCA constructed from the regex are likely to contain a large number of different tokens. Moreover, the size of the DFA for such a regex is usually very large compared to the size of the corresponding NFA. These challenging regexes are usually of a form similar to the ones used in the microbenchmarks (e.g., `.*a.{100}c`).

We measured the average speedup of BVA-Scan against PCRE and RE2 when matching these challenging regexes over adversarial input. For all these regexes, our algorithms from Section 4 are instrumental in reducing the matching time compared to other approaches. Our results show that BVA-Scan offers an average speedup of 151× against PCRE (up to 3660× for large repetition bounds) and of 71× against RE2 (up to 844×), which is considerably larger than the average speedup for the entire dataset (shown in Table 3).

6 RELATED WORK

There is a large number of prior works on ***software regex matchers*** that are based on finite-state automata. Most regex engines use NFAs, DFAs, or some combination of NFAs and DFAs (e.g., RE2 and Hyperscan). PCRE2 [Hazel and Herczeg 2022] is based on backtracking search, but also provides the function *pcre2_dfa_match()* that matches using a DFA. RE2 [RE2 2023] performs an NFA-to-DFA on-the-fly determinization by caching visited states, and switches to an NFA simulation if there are too many DFA states. PCRE2, as opposed to other high-performance regex engines, supports backreferences at the cost of using a backtracking algorithm whose worst-case complexity is exponential in the size of the input. SRM [Saarikivi et al. 2019] is based on Brzozowski derivatives [Brzozowski 1964] (therefore it is similar to DFA-based approaches) extended to support bounded repetition and match extraction.

To speed up matching on large inputs, regex engines often perform pre-filtering by extracting simple string sub-patterns from the regex and applying ***string matching*** algorithms. There are several efficient algorithms for string matching [Aho and Corasick 1975; Boyer and Moore 1977; Commentz-Walter 1979; Karp and Rabin 1987; Knuth et al. 1977]. The popular network intrusion detection system Snort [Snort 2023] uses the Aho-Corasick algorithm [Aho and Corasick 1975]. Grep [Grep 2022] skips parts of the input that cannot match the regex using the Boyer-Moore algorithm [Boyer and Moore 1977]. Hyperscan [Wang et al. 2019] decomposes a regular expression into a collection of strings and other more complex sub-patterns. Hyperscan uses a variant of the shift-or algorithm [Baeza-Yates and Gonnet 1992] to match strings. String matching is further optimized in Hyperscan with a 128-bit SIMD implementation of the shift-or algorithm. Our engine, BVA-Scan, decomposes regexes and uses specialized algorithms (including shift-or and shift-and) for the efficient matching of strings and other simple patterns.

Some regex engines (e.g., RE2) handle bounded repetitions by unfolding them. For example, the regular expression $r\{k\}$ is rewritten into the k-fold concatenation $r \cdot r \cdots r$. The unfolding of bounded repetitions can lead to an exponential blowup in size. It is therefore not surprising that RE2 does not handle regexes of the form $r\{k\}$ for $k > 1000$. Various kinds of ***automata with counters*** were proposed in prior works in order to effectively deal with regexes that contain bounded repetitions (i.e., counting). [Smith et al. 2008] introduce an automaton called XFA, which extends traditional DFA with counters to reduce the memory needed to represent regular expressions with bounded repetitions. Similarly, [Becchi and Crowley 2008] propose an extension to NFA called counting-NFA. [Holík et al. 2019] consider the determinization of a class of counter automata. Their determinization produces smaller automata than the standard determinization into DFAs. [Turoňová et al. 2020] extend the work of [Holík et al. 2019] and present the CA regex engine that handles bounded repetition with a specialized algorithm. They use a class of automata with counters (called “counting automata” or CAs), which are nondeterministic, to represent patterns. CAs are converted into deterministic “counting-set automata” (CsAs). Unfortunately, the CA-to-CsA determinization does not always preserve the semantics, i.e., the language that is recognized may change. Using CsAs, they describe an algorithm that can deal with some cases of bounded repetition efficiently. One problem with the approach of [Turoňová et al. 2020] is that the automata that underlie the matching algorithm sometimes over-approximate the language of the pattern, which can result in incorrect output (a “false positive”, i.e., a false match).

More recently, [Kong et al. 2022] have designed an in-memory hardware architecture that incorporates counters and bit vector modules into the hardware design, thus enabling substantial area and energy savings compared to Micron’s Automata Processor [Dlugosch et al. 2014]. One of the key contributions in [Kong et al. 2022] is a static analysis for regexes that enables the identification of bounded repetitions $r\{m, n\}$ that are easy, in the sense that they can be implemented with a small amount of memory using a single counter of size $\Theta(\log n)$. This static analysis uses nondeterministic counter automata (NCAs) with bounded counters to represent the patterns.

7 CONCLUSION

We have considered the problem of how to efficiently match regular expressions with bounded repetition. Prior work addresses this problem for a more limited class of regexes. We expand this class by considering bounded repetitions of the form $(\sigma_0\sigma_1 \ldots \sigma_{k-1})\{m, n\}$, where each σ_i is a predicate over the alphabet (character class). Then, we continue to further expand to bounded repetitions of the form $r\{m, n\}$, where r is a regex (without bounded repetition) that accepts strings of some fixed length. The vast majority of challenging bounded repetitions that occur in practice fall within the class that we consider and we can therefore handle them efficiently. We have implemented the BVA-Scan regex engine that integrates our efficient algorithms. Our experimental results show that BVA-Scan consistently outperforms RE2, PCRE, and CA on the application benchmarks Snort, Suricata, SpamAssassin, Protomata, ClamAV and RegexLib. As a direction for future work, we plan to integrate the static analysis of [Kong et al. 2022] into BVA-Scan in order to reduce the memory used for handling bounded repetition.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive comments. This research was supported in part by the US National Science Foundation award CCF 2008096 and the Rice University Faculty Initiative Fund.

REFERENCES

Alfred V. Aho and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* 18, 6 (1975), 333–340. https://doi.org/10.1145/360825.360855

Valentin Antimirov. 1996. Partial Derivatives of Regular Expressions and Finite Automaton Constructions. *Theoretical Computer Science* 155, 2 (1996), 291–319. https://doi.org/10.1016/0304-3975(95)00182-4

GNU Awk. 2022. GNU Awk. https://www.gnu.org/software/gawk/ Accessed: March 11, 2023.

Backreferences. 2022. Back Reference in PCRE. https://www.pcre.org/original/doc/html/pcrepattern.html#SEC19 Accessed: March 11, 2023.

Ricardo Baeza-Yates and Gaston H. Gonnet. 1992. A New Approach to Text Searching. *Commun. ACM* 35, 10 (1992), 74–82. https://doi.org/10.1145/135239.135243

Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. 2004. Rule-Based Runtime Verification. In *VMCAI 2004 (LNCS)*, Vol. 2937. Springer, Heidelberg, 44–57. https://doi.org/10.1007/978-3-540-24622-0_5

Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. 2018. Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications. In *Lectures on Runtime Verification: Introductory and Advanced Topics*, Ezio Bartocci and Yliès Falcone (Eds.). LNCS, Vol. 10457. Springer, Cham, 135–175. https://doi.org/10.1007/978-3-319-75632-5_5

Michela Becchi and Patrick Crowley. 2008. Extending Finite Automata to Efficiently Match Perl-Compatible Regular Expressions. In *Proceedings of the 2008 ACM CoNEXT Conference (CoNEXT '08)*. ACM, New York, NY, USA, Article 25, 12 pages. https://doi.org/10.1145/1544012.1544037

Joao Bispo, Ioannis Sourdis, Joao M. P. Cardoso, and Stamatis Vassiliadis. 2006. Regular Expression Matching for Reconfigurable Packet Inspection. In *2006 IEEE International Conference on Field Programmable Technology*. IEEE, USA, 119–126. https://doi.org/10.1109/FPT.2006.270302

Chunkun Bo, Vinh Dang, Elaheh Sadredini, and Kevin Skadron. 2018. Searching for Potential gRNA Off-Target Sites for CRISPR/Cas9 Using Automata Processing Across Different Platforms. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, USA, 737–748. https://doi.org/10.1109/HPCA.2018.00068

Robert S. Boyer and J. Strother Moore. 1977. A Fast String Searching Algorithm. *Commun. ACM* 20, 10 (1977), 762–772. https://doi.org/10.1145/359842.359859

Benjamin C. Brodie, David E. Taylor, and Ron K. Cytron. 2006. A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*. IEEE Computer Society, USA, 191–202. https://doi.org/10.1109/ISCA.2006.7

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494. https://doi.org/10.1145/321239.321249

Agnishom Chattopadhyay and Konstantinos Mamouras. 2020. A Verified Online Monitor for Metric Temporal Logic with Quantitative Semantics. In *RV 2020 (LNCS)*, Jyotirmoy Deshmukh and Dejan Ničković (Eds.), Vol. 12399. Springer, Cham, 383–403. https://doi.org/10.1007/978-3-030-60508-7_21

ClamAV. 2023. ClamAV - Open Source Antivirus Engine. Website. https://www.clamav.net/ Accessed: March 11, 2023.

Beate Commentz-Walter. 1979. A String Matching Algorithm Fast on the Average. In *ICALP 1979 (LNCS)*, Hermann A. Maurer (Ed.), Vol. 71. Springer, Berlin, Heidelberg, 118–132. https://doi.org/10.1007/3-540-09510-1_10

James C. Davis. 2019. 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)*. ACM, New York, NY, USA, 1256–1258. https://doi.org/10.1145/3338906.3342509

Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3088–3098. https://doi.org/10.1109/TPDS.2014.8

Wouter Gelade, Marc Gyssens, and Wim Martens. 2009. Regular Expressions with Counting: Weak versus Strong Determinism. In *MFCS 2009 (LNCS)*, Rastislav Královič and Damian Niwiński (Eds.), Vol. 5734. Springer, Berlin, Heidelberg, 369–381. https://doi.org/10.1007/978-3-642-03816-7_32

Victor Mikhaylovich Glushkov. 1961. The Abstract Theory of Automata. *Russian Mathematical Surveys* 16, 5 (1961), 1–53. https://doi.org/10.1070/RM1961v016n05ABEH004112

Jan Goyvaerts. 2021. Runaway Regular Expressions: Catastrophic Backtracking. https://www.regular-expressions.info/catastrophic.html accessed March 11, 2023.

GNU Grep. 2022. GNU Grep - Global Regular Expression Print. https://www.gnu.org/software/grep/ Accessed: March 11, 2023.

Philip Hazel and Zoltan Herczeg. 2022. PCRE2 - Perl Compatible Regular Expressions v2. https://www.pcre.org/ Accessed: March 11, 2023.

Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Lenka Turoňová, Margus Veanes, and Tomáš Vojnar. 2019. Succinct Determinisation of Counting Automata via Sphere Construction. In *APLAS 2019 (LNCS)*, Anthony Widjaja Lin (Ed.), Vol. 11893.

Springer, Cham, 468–489. https://doi.org/10.1007/978-3-030-34175-6_24

Dag Hovland. 2009. Regular Expressions with Numerical Constraints and Automata with Counters. In *ICTAC 2009 (LNCS)*, Martin Leucker and Carroll Morgan (Eds.), Vol. 5684. Springer, Berlin, Heidelberg, 231–245. https://doi.org/10.1007/978-3-642-03466-4_15

Posix Syntax in PCRE. 2022. Posix Syntax in PCRE. https://www.pcre.org/original/doc/html/pcrepattern.html Accessed: March 11, 2023.

Richard M. Karp and Michael O. Rabin. 1987. Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development* 31, 2 (1987), 249–260. https://doi.org/10.1147/rd.312.0249

Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (1977), 323–350. https://doi.org/10.1137/0206024

Lingkun Kong, Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, Yi Huang, Konstantinos Mamouras, and Kaiyuan Yang. 2022. Software-Hardware Codesign for Efficient In-Memory Regular Pattern Matching. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022).* ACM, New York, NY, USA, 733–748. https://doi.org/10.1145/3519939.3523456

CsA Automata Library. 2021. CsA Automata Library. https://pajda.fit.vutbr.cz/ituronova/countingautomata

Konstantinos Mamouras, Agnishom Chattopadhyay, and Zhifu Wang. 2021a. Algebraic Quantitative Semantics for Efficient Online Temporal Monitoring. In *TACAS 2021 (LNCS)*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.), Vol. 12651. Springer, Cham, 330–348. https://doi.org/10.1007/978-3-030-72016-2_18

Konstantinos Mamouras, Agnishom Chattopadhyay, and Zhifu Wang. 2021b. A Compositional Framework for Quantitative Online Monitoring over Continuous-Time Signals. In *RV 2021 (LNCS)*, Lu Feng and Dana Fisman (Eds.), Vol. 12974. Springer, Cham, 142–163. https://doi.org/10.1007/978-3-030-88494-9_8

Konstantinos Mamouras and Zhifu Wang. 2020. Online Signal Monitoring with Bounded Lag. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3868–3880. https://doi.org/10.1109/TCAD.2020.3013053

Albert R. Meyer and Michael J. Fischer. 1971. Economy of Description by Automata, Grammars, and Formal Systems. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science.* IEEE Computer Society, Los Alamitos, CA, USA, 188–191. https://doi.org/10.1109/SWAT.1971.11

Albert R. Meyer and Larry J. Stockmeyer. 1972. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. In *13th Annual Symposium on Switching and Automata Theory (SWAT 1972).* IEEE Computer Society, Los Alamitos, CA, USA, 125–129. https://doi.org/10.1109/SWAT.1972.29

RE2. 2023. RE2: Google's regular expression library. Website. https://github.com/google/re2 Accessed: March 11, 2023.

RegexLib. 2023. Regular Expression Library. https://regexlib.com/ Accessed: March 11, 2023.

Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration* (Seattle, Washington) *(LISA '99).* USENIX Association, USA, 229–238. https://www.usenix.org/legacy/publications/library/proceedings/lisa99/full_papers/roesch/roesch.pdf

Indranil Roy and Srinivas Aluru. 2016. Discovering Motifs in Biological Sequences Using the Micron Automata Processor. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 13, 1 (2016), 99–111. https://doi.org/10.1109/TCBB.2015.2430313

Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. 2019. Symbolic Regex Matcher. In *TACAS 2019 (LNCS)*, Vol. 11427. Springer, Cham, 372–378. https://doi.org/10.1007/978-3-030-17462-0_24

Christian J. A. Sigrist, Lorenzo Cerutti, Edouard de Castro, Petra S. Langendijk-Genevaux, Virginie Bulliard, Amos Bairoch, and Nicolas Hulo. 2009. PROSITE, A Protein Domain Database for Functional Characterization and Annotation. *Nucleic Acids Research* 38, suppl_1 (2009), D161–D166. https://doi.org/10.1093/nar/gkp885

Randy Smith, Cristian Estan, and Somesh Jha. 2008. XFA: Faster Signature Matching with Extended Automata. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08).* IEEE Computer Society, USA, 187–201. https://doi.org/10.1109/SP.2008.14

Snort. 2023. Snort - Network Intrusion Detection & Prevention System. https://www.snort.org/ Accessed: March 11, 2023.

Apache SpamAssassin. 2022. Apache SpamAssassin. https://spamassassin.apache.org/ Accessed: March 11, 2023.

Larry J. Stockmeyer and Albert R. Meyer. 1973. Word Problems Requiring Exponential Time (Preliminary Report). In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing (STOC '73).* ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/800125.804029

Suricata. 2023. Suricata - Open Source Intrusion Detection and Prevention Engine. https://suricata.io/ Accessed: March 11, 2023.

Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (1968), 419–422. https://doi.org/10.1145/363347.363387

Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. 2020. Regex Matching with Counting-Set Automata. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 218 (2020), 30 pages. https://doi.org/10.1145/3428286

Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-Pattern Regex Matcher for Modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*. USENIX Association, Boston, MA, 631–648. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang

Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. 2006. Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS '06)*. ACM, New York, NY, USA, 93–102. https://doi.org/10.1145/1185347.1185360

Received 2022-10-28; accepted 2023-02-25