Static Analysis for Checking the Disambiguation Robustness of Regular Expressions

KONSTANTINOS MAMOURAS, Rice University, USA
ALEXIS LE GLAUNEC, Rice University, USA
WU ANGELA LI, Rice University, USA
AGNISHOM CHATTOPADHYAY, Rice University, USA

Regular expressions are commonly used for finding and extracting matches from sequence data. Due to the inherent ambiguity of regular expressions, a disambiguation policy must be considered for the match extraction problem, in order to uniquely determine the desired match out of the possibly many matches. The most common disambiguation policies are the POSIX policy and the greedy (PCRE) policy. The POSIX policy chooses the longest match out of the leftmost ones. The greedy policy chooses a leftmost match and further disambiguates using a greedy interpretation of Kleene iteration to match as many times as possible. The choice of disambiguation policy can affect the output of match extraction, which can be an issue for reusing regular expressions across regex engines. In this paper, we introduce and study the notion of disambiguation robustness for regular expressions. A regular expression is robust if its extraction semantics is indifferent to whether the POSIX or greedy disambiguation policy is chosen. This gives rise to a decision problem for regular expressions, which we prove to be PSPACE-complete. We propose a static analysis algorithm for checking the (non-)robustness of regular expressions and two performance optimizations. We have implemented the proposed algorithms and we have shown experimentally that they are practical for analyzing large datasets of regular expressions derived from various application domains.

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

Additional Key Words and Phrases: regex, automata, parsing, disambiguation strategy, static analysis

ACM Reference Format:
Konstantinos Mamouras, Alexis Le Glaunec, Wu Angela Li, and Agnishom Chattopadhyay. 2024. Static Analysis for Checking the Disambiguation Robustness of Regular Expressions. *Proc. ACM Program. Lang.* 8, PLDI, Article 231 (June 2024), 25 pages. https://doi.org/10.1145/3656461

1 INTRODUCTION

Regular expressions are commonly used for searching in text [grep 2024] and for simple text processing [awk 2024; sed 2024]. They have also found applications in numerous domains, including network intrusion detection [Yu et al. 2006], bioinformatics [Roy and Aluru 2016] and runtime verification [Bartocci et al. 2018]. A key computational task is the *membership* problem: Given a regular expression r and a string w as input, does the string belong to the language of the expression? In practice, it is often useful to perform *match extraction*: find and output substrings of the input string that match the desired pattern. For example, one may be interested in extracting all email addresses that appear in a web page. POSIX utilities such as grep can be used to perform such

Authors' addresses: Konstantinos Mamouras, Rice University, Houston, USA, mamouras@rice.edu; Alexis Le Glaunec, Rice University, Houston, USA, alexis.leglaunec@rice.edu; Wu Angela Li, Rice University, Houston, USA, awl@rice.edu; Agnishom Chattopadhyay, Rice University, Houston, USA, agnishom@rice.edu.

ACM 2475-1421/2024/6-ART231
https://doi.org/10.1145/3656461

computations. All mainstream programming languages come equipped with libraries for regular expression matching. Due to the inherent ambiguity of regular expressions, the match extraction problem requires a *disambiguation policy*, which describes how a match should be chosen when there are many possible choices. The POSIX standard imposes the “leftmost longest” rule, i.e, it chooses the match that starts as far left as possible that cannot be extended further to the right. The PCRE policy also prefers a leftmost match. But it further disambiguates by favoring the most *greedy* match. For a Kleene iteration r^*, it prefers repeating r as many times as possible. Moreover, for a nondeterministic choice $r_1 \mid r_2$, it prefers the left choice r_1 over the right choice r_2. Thus, given the string `aab` and the regular expression `(a|ab)*`, the POSIX match is the entire string but the PCRE match is the prefix `aa`. The greedy disambiguation policy naturally arises from the implementation of backtracking engines. To the best of our knowledge, all backtracking-based engines follow the PCRE matching semantics. Cox [2010] discusses how the greedy semantics can be implemented with a backtracking-free automata-based algorithm.

The reuse of regular expressions is common in practice. When programmers consider using a regular expression for a certain task, it is typical for them to reuse a regex that is already written by another programmer or contained in a curated list of regexes [Michael et al. 2019]. Hodován et al. [2010] have found that in several popular web sites, only about 4% of the regular expressions are unique. Wang et al. [2019] have observed that once written, 95% of regular expressions used in GitHub projects are not edited in the future. Davis et al. [2019] have presented an empirical study about the various syntactic differences in regular expressions across several different libraries and engines. The reuse of regular expression raises potential issues of portability across regex engines.

In this paper, we address the issue of reusing regular expressions across regex engines that use different disambiguation policies (POSIX versus greedy/PCRE). This problem is not just a theoretical curiosity. We have investigated several datasets of regular expressions (Snort, Suricata, SpamAssassin, and RegexLib) that cover various application domains and we have found that there exist many regular expressions that do not have the same match extraction semantics when the disambiguation policy is changed from POSIX to greedy (or vice versa). For example, the regex `((\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5])\.){3}(\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5])` (which is taken from the Snort dataset) detects a range of IP addresses typically used by trojans for a data exfiltration attack from an infected target. For the input `HOST: 239.255.255.250` which is taken from a real PCAP file, the greedy semantics would return the match `239.255.255.2` whereas the complete IP address is `239.255.255.250`. If one wants to extract the match to compare against a list of well-known suspicious IP addresses, an alert can be erroneously ignored or triggered depending on the disambiguation semantics. So, the difference in semantics can have security implications.

Main Contributions. We make the following contributions in this paper:

(1) We introduce the notion of *(disambiguation) robustness* for regular expressions. A regular expression is robust if, for every input string, the preferred match according to the greedy disambiguation policy (“leftmost greedy”) is the same as the preferred match according to the POSIX disambiguation policy (“leftmost longest”). This gives rise to the computational problem of deciding whether a regular expression is robust or not. We prove that this problem is PSPACE-hard. If backreferences are allowed, then the problem becomes undecidable.

(2) Using a variant of classical ε-NFAs that indicate priorities on the transitions (when more than one transition emanates from a state), we characterize non-robustness in terms of a reachability property in a product graph of ε-NFA configuration pairs. In these configuration pairs, one configuration is for a “greedy” execution of the automaton and one configuration for a parallel “POSIX” execution. This establishes that the problem of checking robustness is contained in PSPACE and is therefore PSPACE-complete. The characterization also gives rise to a static

$$\begin{aligned}
w, [i, j] &\models \varepsilon \iff i = j \\
w, [i, j] &\models \sigma \iff j = i + 1 \text{ and } w(i) \in \sigma \\
w, [i, j] &\models r_1 \mid r_2 \iff w, [i, j] \models r_1 \text{ or } w, [i, j] \models r_2
\end{aligned}
\qquad
\begin{aligned}
w, [i, j] \models r_1 \cdot r_2 &\iff \text{there is } k \text{ with } i \leq k \leq j \text{ s.t.} \\
& w, [i, k] \models r_1 \text{ and } w, [k, j] \models r_2 \\
w, [i, j] \models r^* &\iff i = j \text{ or there is } k \text{ with } i < k \leq j \\
& \text{s.t. } w, [i, k] \models r \text{ and } w, [k, j] \models r^*
\end{aligned}$$

Fig. 1. Formal semantics of regular expressions. The *satisfaction relation* $\models$ relates a string $w \in \Sigma$, a location $[i, j]$ with $0 \leq i \leq j \leq |w|$, and a regular expression r.

analysis algorithm for (non-)robustness that explores the graph of configuration pairs and returns a witness for non-robustness (when one exists).

(3) We identify two performance optimizations for dealing with the computationally difficult problem of robustness checking. The first optimization relies on the notion of *end-unambiguity*, which says that a match cannot be further extended to the right. The second optimization relies on several properties about the preservation of robustness when right-concatenating some simple (but commonly occuring) regular expressions.

(4) We have implemented the proposed static analysis algorithm for (non-)robustness checking, including the two aforementioned optimizations. To the best of our knowledge, this is the first tool that performs this static analysis of regular expressions. Using our tool, we identify hundreds of regular expressions in real regex datasets that are not robust (and are therefore potentially problematic for reuse). We also show that non-robustness manifests when real-world input strings are used. Finally, we establish experimentally that our tool can analyze thousands of regular expressions in a reasonable amount of time and that the optimizations offer a substantial performance improvement. Our most optimized algorithm analyzes a regular expression in less than 20 msec on average (over the datasets that we have considered).

2 DISAMBIGUATION POLICIES

In this section, we provide formal definitions for the greedy and POSIX disambiguation policies for match extraction. Instead of using parse trees, we define by induction the greedy preference order on the set of all matches of a regular expression. Using the greedy and POSIX preference orders, we formally define the concept of (disambiguation) robustness. Intuitively, a regular expression is robust if, for every input string, the most preferred match specified by the greedy policy is the same as the one specified by the POSIX policy. This notion gives rise to the computational problem of checking whether a regular expression is (non-)robust, which we show to be PSPACE-hard.

Let Σ be a finite alphabet of symbols (letters, characters). A predicate $\sigma \subseteq \Sigma$ is called a *character class*. The set $\mathrm{Reg}(\Sigma)$ of *regular expressions* (regexes) is defined by the following grammar: $r, r_1, r_2 ::= \varepsilon \mid \sigma \mid (r_1 \mid r_2) \mid r_1 \cdot r_2 \mid r^*$. Concatenation is also written as $r_1 r_2$ to reduce notational clutter. The notation r^+ ("repetition of r at least once") is abbreviation for rr^*. The notation $r?$ is abbreviation for $r \mid \varepsilon$. For a regular expression r, the notation r^n is abbreviation for the concatenation $r \cdot r \cdots r$ (n times). The notation $r\{n\}$ is also commonly used to describe the repetition of r exactly n times. More generally, we write $r\{m, n\} = r^m (r?)^{n-m}$ to denote the repetition of r from m to n times. Every regular expression r denotes a language $\mathcal{L}(r) \subseteq \Sigma^*$, defined in the standard way.

We write $|w|$ to denote the length of a string w. The empty string (i.e., the string of length 0) is denoted by ε. For a string $w \in \Sigma^*$, we will call a pair $[i, j]$ with $0 \leq i \leq j \leq |w|$ a *location* in w. A *position* in $|w|$ is an index in the range $0, 1, \ldots, |w|$. We write $w[i..j]$ for the substring of w at location $[i, j]$. E.g., for the string $w = abbcabab$ (length $|w| = 8$), we have that $w[0..3] = abb$, $w[1..5] = bbca$, and $w[4..7] = aba$, and $w[5..8] = bab$.

We can also think of a string $w \in \Sigma^*$ as a function from $\mathrm{dom}(w) = \{0, 1, \ldots, n-1\}$ to Σ, where $n = |w|$ and $w(i)$ is the letter at position i. This means that $w = w(0)w(1) \ldots w(n-1)$.

Definition 1 (Formal Match Semantics). Let $w \in \Sigma^*$ be a string, i and j be integers satisfying $0 \leq i \leq j \leq |w|$, and $r \in \mathrm{Reg}(\Sigma)$ be a regular expression. The relation $\models$ is defined by induction as shown in Fig. 1. The *match-set* $\mathcal{M}(w, r, i)$ for a regex r and a word w at position $i \in \{0, \dots, |w|\}$ is the set of locations with left endpoint i where r has a match in w, i.e., $\mathcal{M}(w, r, i) = \{[i, j] \mid 0 \leq i \leq j \leq |w| \text{ and } w, [i, j] \models r\}$. The *match-set* $\mathcal{M}(w, r)$ for a regex r and a word w is the set of all locations where r has a match in w, i.e., $\mathcal{M}(w, r) = \{[i, j] \mid 0 \leq i \leq j \leq |w| \text{ and } w, [i, j] \models r\}$. Alternatively, this can be defined as $\mathcal{M}(w, r) = \bigcup\{\mathcal{M}(w, r, i) \mid 0 \leq i \leq |w|\}$.

A *decomposition* of a location $[i, j]$ (where $i \leq j$) is a nonempty finite sequence of locations $[i_1, j_1], [i_2, j_2], \dots, [i_n, j_n]$ with $i_1 = i$, $j_n = j$, and $j_k = i_{k+1}$ for every $k = 1, 2, \dots, n-1$. We note that $w, [i, j] \models r^*$ iff $i = j$ or there exists a decomposition $[i_1, j_1], [i_2, j_2], \dots, [i_n, j_n]$ of $[i, j]$ such that $w, [i_k, j_k] \models r$ for every $k = 1, \dots, n$.

Let r be a regular expression, $w \in \Sigma^*$, and $[i, j]$ be a location in w. It holds that $w[i..j] \in \mathcal{L}(r)$ iff $w, [i, j] \models r$. This means that $\models$ gives us an alternative way to look at the semantics of regular expressions that is more flexible than the usual definition using languages.

2.1 Disambiguation

We will define the *greedy preference order* $<_i^r$ on the match-set $\mathcal{M}(w, r, i)$. The preference order $<_i^r$ is a linear order. The definition is by induction on the structure of r. For the base case $r = \varepsilon$, we have that $\mathcal{M}(w, \varepsilon, i) = \{[i, i]\}$, and $<_i^r$ is defined to be empty. For the case $r = \sigma \subseteq \Sigma$, $\mathcal{M}(w, \sigma, i) = \{[i, i+1] \mid w(i) \in \sigma\}$. $\mathcal{M}(w, \sigma, i)$ is either empty or singleton, and we define $<_i^r$ to be empty.

Consider now the case $r = r_1 \mid r_2$ of nondeterministic choice. Define $T_i = \{\mathsf{inl}[i, j] \mid w, [i, j] \models r_1\} \cup \{\mathsf{inr}[i, j] \mid w, [i, j] \models r_2\}$. The "flattening" function $\rho_i : T_i \to \mathcal{M}(w, r)$ is given by $\rho_i(\mathsf{inl}[i, j]) = \rho_i(\mathsf{inr}[i, j]) = [i, j]$. The order $<_i$ on T_i is generated by the rules:

$$\frac{[i, j] <_i^{r_1} [i, j']}{\mathsf{inl}[i, j] <_i \mathsf{inl}[i, j']} \qquad \frac{[i, j] <_i^{r_2} [i, j']}{\mathsf{inr}[i, j] <_i \mathsf{inr}[i, j']} \qquad \mathsf{inl}[i, j] <_i \mathsf{inr}[i, j']$$

Finally, we define $[i, j] <_i^r [i, j']$ iff $\min \rho_i^{-1}([i, j]) <_i \min \rho_i^{-1}([i, j'])$, where the *min* operator is with respect to the linear order $<_i$.

For the case $r = r_1 r_2$ of concatenation, define $T_i = \{[i, j][j, k] \mid w, [i, j] \models r_1 \text{ and } w, [j, k] \models r_2\}$. The "flattening" function $\rho_i : T_i \to \mathcal{M}(w, r)$ is given by $\rho_i([i, j][j, k]) = [i, k]$. The order $<_i$ on T_i is generated by the following rules:

$$\frac{[i, j] <_i^{r_1} [i, j']}{[i, j][j, k] <_i [i, j'][j', k']} \qquad \frac{[j, k] <_j^{r_2} [j, k']}{[i, j][j, k] <_i [i, j][j, k']}$$

Finally, we define $[i, j] <_i^r [i, j']$ iff $\min \rho_i^{-1}([i, j]) <_i \min \rho_i^{-1}([i, j'])$, where the *min* operator is with respect to the linear order $<_i$.

Now, we deal with the case $r = r_1^*$ of Kleene iteration. We consider the sequences of locations

$$\mathcal{S}_i = \{\langle\rangle\} \cup \{\langle [i_1, i_2], [i_2, i_3], \dots, [i_n, i_{n+1}] \rangle \mid n \geq 1, i = i_1, \text{ and } i_{k+1} - i_k \geq 1 \text{ for all } k = 1, \dots, n\}.$$

Define $T_i = \{S \in \mathcal{S}_i \mid w, [i_k, i_{k+1}] \models r_1 \text{ for every location } [i_k, i_{k+1}] \text{ in } S\}$ and the "flattening" function $\rho_i : T_i \to \mathcal{M}(w, r)$ by $\rho_i(\langle [i_1, i_2], \dots, [i_n, i_{n+1}] \rangle) = [i_1, i_{n+1}]$ and $\rho_i(\langle\rangle) = [i, i]$. The order $<_i$ on T_i is generated by the following rules:

$$\frac{[i, j] <_i^{r_1} [i, j']}{[i, j] \cdot S <_i [i, j'] \cdot S'} \qquad \frac{S <_j S'}{[i, j] \cdot S <_i [i, j] \cdot S'} \qquad S <_i \langle\rangle \text{ when } S \neq \langle\rangle$$

We define $[i, j] <_i^r [i, j']$ iff $\min \rho_i^{-1}([i, j]) < \min \rho_i^{-1}([i, j'])$, where the *min* operator is with respect to the linear order $<_i$.

Finally, we define the *greedy preference order* $<^r$ on the match-set $\mathcal{M}(w, r)$ as follows: $[i, j] <^r [i', j']$ iff $i < i'$ or ($i = i'$ and $[i, j] <^r_i [i, j']$).

Example 2 (Greedy Preference Order)**.** Let us consider the example $r =$ `b+` and $w =$ `abbbabb`. The match-set for r in w is $\mathcal{M}(w, r) = \{[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4], [5, 6], [5, 7], [6, 7]\}$ and the greedy preference order is $[1, 4] < [1, 3] < [1, 2] < [2, 4] < [2, 3] < [3, 4] < [5, 7] < [5, 6] < [6, 7]$. The most preferred match of r in w is at location $[1, 4]$ (leftmost greedy match).

The *POSIX preference order* $\prec$ on match locations is defined by $[i, j] \prec [i', j']$ iff $i < i'$ or ($i = i'$ and $j > j'$). The intuition is that the POSIX disambiguation policy prefers the longest out of the leftmost matches (also referred to as "leftmost longest").

Example 3 (Greedy vs POSIX Preference)**.** For $r =$ `a.*b|a.*c` and $w =$ `baacaabc`, the match-set is $\mathcal{M}(w, r) = \{[1, 4], [1, 7], [1, 8], [2, 4], [2, 7], [2, 8], [4, 7], [4, 8], [5, 7], [5, 8]\}$. The greedy preference order is $[1, 7] < [1, 8] < [1, 4] < [2, 7] < [2, 8] < [2, 4] < [4, 7] < [4, 8] < [5, 7] < [5, 8]$. The POSIX preference order is $[1, 8] \prec [1, 7] \prec [1, 4] \prec [2, 8] \prec [2, 7] \prec [2, 4] \prec [4, 8] \prec [4, 7] \prec [5, 8] \prec [5, 7]$. The most preferred greedy (resp., POSIX) match is at location $[1, 7]$ (resp., $[1, 8]$).

2.2 Robustness and Computational Hardness

The most common computational problem associated with regular expressions is the matching problem. In the case of Boolean matching (does there exist a match?), there is no need for a disambiguation policy. For match extraction, a disambiguation policy has to be specified.

- *Greedy Matching*: Given a regular expression $r \in \mathrm{Reg}(\Sigma)$ and a string $w \in \Sigma^*$, output the most preferred greedy match (i.e., leftmost greedy) of r in w (if one exists).
- *POSIX Matching*: Given a regular expression $r \in \mathrm{Reg}(\Sigma)$ and a string $w \in \Sigma^*$, output the most preferred POSIX match (i.e., leftmost longest) of r in w (if one exists).

Definition 4 (Denotations and Robustness). Let $r \in \mathrm{Reg}(\Sigma)$. The *greedy denotation* of r is a function $[\![r]\!]_{\mathrm{G}} : \Sigma^* \to \mathrm{Option}(\mathbb{N} \times \mathbb{N})$ defined as follows: $[\![r]\!]_{\mathrm{G}}(w) = \mathrm{None}$ if $\mathcal{M}(w, r) = \emptyset$, and $[\![r]\!]_{\mathrm{G}}(w) = \mathrm{Some}(\min \mathcal{M}(w, r))$ if $\mathcal{M}(w, r) \neq \emptyset$, where the minimum is taken with respect to the greedy preference order. The *POSIX denotation* $[\![r]\!]_{\mathrm{P}} : \Sigma^* \to \mathrm{Option}(\mathbb{N} \times \mathbb{N})$ of r is defined similarly, with the difference being that the minimum is taken with respect to the POSIX preference order.

We say that a regular expression r is ***(disambiguation) robust*** (i.e., robust with respect to the choice of disambiguation policy) if the most preferred match in any string is the same regardless of whether the greedy or POSIX policy is used for disambiguation. In other words, r is defined to be robust iff its greedy and POSIX denotations are equal, i.e., $[\![r]\!]_{\mathrm{G}} = [\![r]\!]_{\mathrm{P}}$.

Robustness is a property over regular expressions, so it gives rise to a corresponding decision computational problem (*robustness analysis*). Define the IsRobust problem as follows: Given a regular expression $r \in \mathrm{Reg}(\Sigma)$, is r robust (in the sense of Definition 4)?

Theorem 5 (Hardness of Checking Robustness). The problem IsRobust is PSPACE-hard.

Proof. Recall that a regex r is called *universal* if $\mathcal{L}(r) = \Sigma^*$. We will reduce the universality problem for regular expressions, which is known to be PSPACE-complete, to the problem IsRobust. Let Σ be the alphabet for the input expression. Suppose that $\triangleright$ ("left marker") and $\triangleleft$ ("right marker") are symbols that are not in Σ. Define the alphabet $\Gamma = \Sigma \cup \{\triangleright, \triangleleft\}$. The function $f : \mathrm{Reg}(\Sigma) \to \mathrm{Reg}(\Gamma)$ is defined by $f(r) = \triangleright(r\triangleleft)? \mid \triangleright\Sigma^*\triangleleft$. Notice that $\mathcal{L}(f(r)) = \{\triangleright\} \cup \{\triangleright w \triangleleft \mid w \in \Sigma^*\}$. We claim that, for every $r \in \mathrm{Reg}(\Sigma)$, r is universal iff $f(r)$ is robust.

Suppose that r is universal. Let w be an arbitrary string over $\Sigma \cup \{\triangleright, \triangleleft\}$. We will show that the greedy and POSIX preferred matches for $f(r)$ in w are the same. If w does not contain $\triangleright$, then there is no match and we have agreement. We can assume from now on that w contains $\triangleright$. Suppose that the first occurrence of $\triangleright$ is at location $[i, i+1]$. There can be no match starting earlier than i, so the preferred match for both the greedy and POSIX policy must start at position i (both policies want the leftmost match). If the suffix $w[i+1..]$ contains no occurrence of $\triangleleft$, then the only match is $\triangleright$ at location $[i, i+1]$, so there is agreement. Now, we consider the case where $w[i+1..]$ contains at least one occurrence of $\triangleleft$ and the earliest occurrence is at location $[j, j+1]$ with $j \geq i+1$. There can be no match $[i, j']$ with $j' > j+1$, so the only matches are at locations $[i, i+1]$ and $[i, j+1]$. The POSIX policy prefers the match at $[i, j+1]$ because it is longer. The greedy policy will choose to match using $\triangleright(r\triangleleft)?$ instead of $\triangleright\Sigma^*\triangleleft$, but it will also prefer the match at location $[i, j+1]$ because $r\triangleleft$ matches at $[i+1, j+1]$ (since r is universal). So, there is agreement in all cases.

Now, we assume that r is not universal, which means that there is a string $v \in \Sigma^*$ such that $v \notin \mathcal{L}(r)$. Consider the string $w = \triangleright v \triangleleft$. According to the POSIX policy, the most preferred match is at location $[0, |w|]$ (entire string), because it is the longest one. The greedy policy, on the other hand, prefers the match at location $[0, 1]$, because it favors the choice $\triangleright(r\triangleleft)?$ over $\triangleright\Sigma^*\triangleleft$. Note that $r\triangleleft$ does not match at location $[1, |w|]$, because $w[1..|w|-1] = v$.

We have established that r is universal iff $f(r)$ is robust. The function f can be computed in polynomial time. So, IsRobust is PSPACE-hard. □

The proof of Theorem 5 gives a reduction from universality to IsRobust that works for any class of regexes that allows the construction $f(r)$. So, for regular expressions with backreferences, IsRobust is at least as hard as universality, which is undecidable [Freydenberger 2013]. It would be interesting to study the complexity of IsRobust when lookaround assertions [Mamouras and Chattopadhyay 2024] and bounded repetition [Kong et al. 2022; Le Glaunec et al. 2023; Wen et al. 2024] are allowed. These constructs make regular expressions more succinct.

3 GREEDY NONDETERMINISTIC FINITE AUTOMATA

In this section, we consider variants of classical NFAs that are appropriate for greedy matching, which we call greedy NFAs or GNFAs. A GNFA can have ε-transitions. The GNFA model is a convenient translation target for regular expressions. The main feature of this model of automata is that it assigns priorities to transitions when there is a choice to be made.

Informally, a "greedy NFA" (GNFA) is an ε-NFA $\mathcal{A}$ that satisfies the following properties: (1) It has states $Q = \{0, 1, \ldots, m-1\}$, where m is the size of $\mathcal{A}$ (i.e., number of states). (2) It has a unique initial state, which is always state 0. (3) It has a unique final state, which is always state $m-1$. (4) The final state has no successors, i.e., no transitions emanating from it. (5) Every non-final state is of one of three types: a "guarded" state, or a "jump" state, or a "(nondeterministic) choice" state. A guarded state q has the unique successor $q' = q+1$ and the transition $q \longrightarrow q'$ is labeled with some character class $\sigma \subseteq \Sigma$. A jump state q has a unique successor q' and the transition $q \longrightarrow q'$ is labeled with ε. A choice state q has exactly two successors $q' < q''$. The transition $q \longrightarrow q'$ is labeled with ε / $\mathtt{0}$ and the transition $q \longrightarrow q''$ is labeled with ε / $\mathtt{1}$. Since a choice state has two outgoing transitions, the intuition is that the one labeled with ε / $\mathtt{0}$ is preferred over the one labeled with ε / $\mathtt{1}$. We write $\mathbb{D} = \{\mathtt{0}, \mathtt{1}\}$ for the set of labels that determine the choice ("$\mathbb{D}$" stands for direction).

Definition 6. A ***greedy NFA*** (with ε-transitions) or ***GNFA*** of size m over the alphabet Σ is a tuple $\mathcal{A} = (Q, \Delta)$, where $Q = \{0, 1, \ldots, m-1\}$ is the *set of states*, 0 is the *initial state*, $m-1$ is the *final state*, and $\Delta : \{0, 1, \ldots, m-2\} \to \mathcal{P}(\Sigma) \oplus Q \oplus (Q \times Q)$ is the *transition function* (where $\oplus$ is the operation of disjoint union, i.e., coproduct). We also write $|\mathcal{A}|$ for the size of $\mathcal{A}$.

If $\Delta(q) = \sigma$ for some character class σ, then q is called a *guarded state* (and its successor is $q' = q+1$). If $\Delta(q) = q'$ for some state $q' \in Q$, then q is called a *jump state*. If $\Delta(q) = (q', q'')$ for states $q', q'' \in Q$, then q is called a *(nondeterministic) choice state* and it must hold that $q' < q''$.

Suppose that q is a state of a GNFA. We write $q \to^{\sigma} q+1$ to indicate that $\Delta(q) = \sigma$, where $q+1$ is the unique successor of the guarded state q. Moreover, for a symbol $a \in \Sigma$, $q \to^{a} q+1$ indicates that $a \in \Delta(q)$. Similarly, we write $q \to^{\varepsilon} q'$ to indicate that $\Delta(q) = q'$. Finally, we write $q \to^{\varepsilon/\mathtt{0}} q'$ and $q \to^{\varepsilon/\mathtt{1}} q''$ to indicate that $\Delta(q) = (q', q'')$. A *path* in $\mathcal{A}$ is a sequence $q_0 \to^{x_0} q_1 \to^{x_1} q_2 \to^{x_2} \cdots \to^{x_{n-1}} q_n$ where (1) each x_i is either a symbol $a \in \Sigma$ or one of $\varepsilon, \varepsilon\ /\ \mathtt{0}, \varepsilon\ /\ \mathtt{1}$, and (2) $q_i \to^{x_i} q_{i+1}$ for every $i = 0, \dots, n-1$. If the path π_1 ends at state q and the path π_2 starts from q, then $\pi_1 \diamond \pi_2$ is the path that results from concatenating π_1 and π_2 (fusing the states q at the boundary). The *(symbol) label* of a path π, denoted $\mathsf{label}(\pi)$, is the sequence of all alphabet symbols seen in it from left to right. The *disambiguation trace* $\mathrm{trc}(\pi)$ of a path π is the sequence of choice labels $\mathtt{0}, \mathtt{1}$ seen in it from left to right. An *accepting path* in $\mathcal{A}$ is a path whose first state is the initial state and whose last state is the final state. We say that $\mathcal{A}$ *accepts* a string $w \in \Sigma^*$ if there is an accepting path in $\mathcal{A}$ whose label is equal to w.

The Thompson construction is a well-known way to construct ε-NFAs that implement regular expressions. Despite the similarity, we provide the formal definition for GNFAs below, since the particular definitions that we use here are important for the algorithms that will be presented later.

Definition 7 (Thompson Construction). For the regex ε, we define $\mathcal{A}_\varepsilon$ to be the GNFA with $Q = \{0\}$, where the unique state is both initial and final. For a *character class* σ, we define $\mathcal{A}_\sigma$ to have states $Q = \{0, 1\}$ and $\Delta(0) = \sigma$. Let $\mathcal{A}_1 = (Q_1, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \Delta_2)$ be GNFAs with sizes $m_1 = |Q_1|$ and $m_2 = |Q_2|$ respectively. We define the *concatenation* $\mathcal{A}_1 \cdot \mathcal{A}_2$ to be the GNFA (Q, Δ) of size $m = |Q| = (m_1 - 1) + m_2$ where $\Delta(q) = \Delta_1(q)$ if $0 \le q < m_1 - 1$ and

$$\Delta(q) = (m_1 - 1) + \Delta_2(q - (m_1 - 1)), \text{ if } m_1 - 1 \le q < m_1 + m_2 - 2.$$

Informally, the concatenation collapses the final state of $\mathcal{A}_1$ with the initial state of $\mathcal{A}_2$. The *(nondeterministic) choice* $\mathcal{A}_1 \mid \mathcal{A}_2$ is the GNFA (Q, Δ) of size $m = |Q| = 1 + m_1 + m_2$, given by

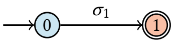
$$\begin{aligned} \Delta(0) &= (1, 1 + m_1) & \qquad \Delta(q) &= 1 + \Delta_1(q - 1), \text{ if } 1 \le q < m_1 \\ \Delta(m_1) &= m_1 + m_2 & \qquad \Delta(q) &= (m_1 + 1) + \Delta_2(q - (m_1 + 1)), \text{ if } m_1 + 1 \le q < m_1 + m_2 \end{aligned}$$

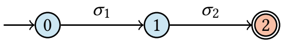
The *(Kleene) iteration* $\mathcal{A}_1^*$ is the GNFA (Q, Δ) of size $m = |Q| = 1 + m_1 + 1$, defined as follows:

$$\Delta(0) = (1, 1 + m_1) \qquad \Delta(q) = 1 + \Delta_1(q - 1), \text{ if } 1 \le q < m_1 \qquad \Delta(m_1) = 0$$

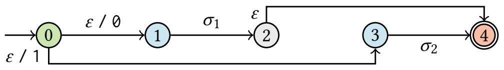
For a regular expression r, the Thompson automaton $\mathcal{A}_r$ results from r by applying each of the above constructors for the corresponding regular operator.

Example 8 (Thompson Construction). The Thompson GNFA for σ has states $Q = \{0, 1\}$ with $\Delta(0) = \sigma$. The Thompson GNFA for $\sigma_1\sigma_2$ has states $Q = \{0, 1, 2\}$ with $\Delta(0) = \sigma_1$ and $\Delta(1) = \sigma_2$.

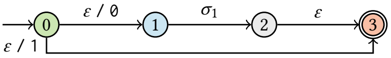


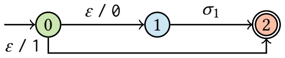


Notice that guarded states are colored with blue. A final state is colored with orange. The Thompson GNFA for $\sigma_1 \mid \sigma_2$ has states $Q = \{0, 1, 2, 3, 4\}$ with $\Delta(0) = (1, 3)$, $\Delta(1) = \sigma_1$, $\Delta(2) = 4$, and $\Delta(3) = \sigma_2$.

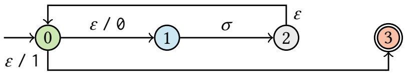


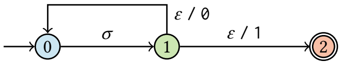
Choice states are colored with green and jump states are colored with gray. The transition labeled with $\varepsilon\ /\ \mathtt{0}$ is of higher priority than the transition labeled with $\varepsilon\ /\ \mathtt{1}$. This encodes the fact that the greedy policy prefers matching σ_1 over matching σ_2. The Thompson GNFA for $\sigma? = \sigma \mid \varepsilon$ has states $Q = \{0, 1, 2, 3\}$ with $\Delta(0) = (1, 3)$, $\Delta(1) = \sigma$, and $\Delta(2) = 3$ (see diagram below on the left). Since states 2 and 3 are connected with an ε-transition, they can be collapsed. This streamlines the GNFA construction for the regex σ? (shown below on the right).





The Thompson GNFA for σ^* has states $Q = \{0,1,2,3\}$ with $\Delta(0) = (1,3)$, $\Delta(1) = \sigma$, and $\Delta(2) = 0$ (see diagram below on the left). For the regex σ^+, we can construct a GNFA that has fewer states than what we would get with the encoding $r^+ = rr^*$ (see diagram below on the right).





The transition labeled with ε / 0 indicates that it is preferred to repeat the loop rather than end it. This is consistent with the greedy (also called "eager") interpretation of Kleene iteration.

We use the *lexicographic order* $<$ on elements of $\mathbb{D}^*$. The order $<$ is linear (i.e., total). For $\tau, \tau' \in \mathbb{D}^*$, we define $\tau \ll \tau'$ iff there exists $i \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau')$ such that $\tau[0..i] = \tau'[0..i]$ and $\tau(i) < \tau'(i)$ (i.e., $\tau(i) = 0$ and $\tau'(i) = 1$). Notice that $\tau \ll \tau'$ implies $\tau < \tau'$. For all $\tau, \tau', \rho, \rho' \in \mathbb{D}^*$, $\tau \ll \tau'$ implies $\tau\rho \ll \tau'\rho'$. Moreover, $\tau < \tau'$ iff $\tau \ll \tau'$ or (τ is a strict prefix of τ'). If τ is a strict prefix of τ', then τ and τ' are not comparable with respect to the $\ll$ order.

Definition 9 (Epsilon Paths). Let $\mathcal{A}$ be a GNFA and $q, q' \in Q$. We define $\mathsf{P}_\varepsilon(q, q')$ to contain the acyclic ε-paths π from q to q' (i.e., $\mathsf{label}(\pi) = \varepsilon$). We also define $\mathsf{R}_\varepsilon(q) = \{q' \mid \mathsf{P}_\varepsilon(q, q') \neq \emptyset\}$. That is, $\mathsf{R}_\varepsilon(q)$ is the set of states that are ε-reachable from q. Finally, $\mathsf{T}_\varepsilon(q, q') = \min\{\mathsf{trc}(\pi) \mid \pi \in \mathsf{P}_\varepsilon(q, q')\}$ is the least ("best") trace of the acyclic ε-paths from q to q' (min is taken w.r.t. $<$).

A *(greedy) priority* is a pair $(i, \tau) \in \mathbb{N} \times \mathbb{D}^*$. We define the order $<$ on priorities as follows: $(i, \tau) < (i', \tau')$ iff $i < i'$ or ($i = i'$ and $\tau < \tau'$). Similarly, $(i, \tau) \ll (i', \tau')$ iff $i < i'$ or ($i = i'$ and $\tau \ll \tau'$). For a priority $(i, \tau) \in \mathbb{N} \times \mathbb{D}^*$ and $\tau' \in \mathbb{D}^*$, we define the concatenation $(i, \tau) \cdot \tau' = (i, \tau\tau')$.

In Definition 10, we will also consider triples $(i, j, \tau) \in \mathbb{N} \times \mathbb{N} \times \mathbb{D}^*$. Informally, we order these triples and we also compare them with priorities by dropping the j component: $(i, \tau), (i, j, \tau) < (i', \tau'), (i', j', \tau')$ iff $(i, \tau) < (i', \tau')$ and $(i, \tau), (i, j, \tau) \ll (i', \tau'), (i', j', \tau')$ iff $(i, \tau) \ll (i', \tau')$. We also define the relation $\simeq$ as follows: $(i, \tau), (i, j, \tau) \simeq (i', \tau'), (i', j', \tau')$ iff $(i, \tau) = (i', \tau')$.

Definition 10 (Greedy Configuration). Let $\mathcal{A}$ be a GNFA with initial state $q_{in} = 0$ and final state $q_{fin} = |\mathcal{A}| - 1$. A path is said to be *$\varepsilon$-acyclic* if it does not contain any ε-cycle. For $w \in \Sigma^*$ and $0 \leq i \leq j \leq |w|$, we define the set $\textsc{Paths}(w, i, j, q)$ of all ε-acyclic paths π in $\mathcal{A}$ from q_{in} to q with $\mathsf{label}(\pi) = w[i..j]$. Moreover, $\textsc{Paths}(w, q) = \bigcup_{i=0}^{|w|} \textsc{Paths}(w, i, |w|, q)$ and $\textsc{Reach}(w) = \{q \mid \textsc{Paths}(w, q) \neq \emptyset\}$. If $q \in \textsc{Reach}(w)$, we define the *greedy priority*

$$\mathsf{GPr}(w, q) = \min\{(i, \mathsf{trc}(\pi)) \mid 0 \leq i \leq |w| \text{ and } \pi \in \textsc{Paths}(w, i, |w|, q)\}.$$

The set of *accepting paths* for w (together with their start and end positions) is

$$\textsc{Acc}(w) = \{(i, j, \pi) \mid 0 \leq i \leq j \leq |w| \text{ and } \pi \in \textsc{Paths}(w, i, j, q_{fin})\}.$$

We define $\textsc{Matched}(w) = 1$ if $\textsc{Acc}(w) \neq \emptyset$ and $\textsc{Matched}(w) = 0$ if $\textsc{Acc}(w) = \emptyset$. The *best greedy match* for w is $\textsc{GBest}(w) = \min\{(i, j, \mathsf{trc}(\pi)) \mid (i, j, \pi) \in \textsc{Acc}(w)\}$. The $<$ order is not total on triples, but the min here is well-defined because the trace $\mathsf{trc}(\pi)$ uniquely determines j in a GNFA.

The *greedy configuration* for w is a partial map $M = \textsc{GCfg}(w)$ whose domain contains the guarded states q of $\textsc{Reach}(w)$ that satisfy $\mathsf{GPr}(w, q) < \textsc{GBest}(w)$. The configuration maps a state q to its priority, that is, $M(q) = \mathsf{GPr}(w, q)$ for every $q \in \mathrm{dom}(M)$.

We write $\mathbb{S}$ for the set of all state identifiers (i.e., $\mathbb{S} = \mathbb{N}$, but $\mathbb{S}$ helps distinguish the use of numbers as states). We can represent $\text{GC}\textsc{fg}(w)$ as a vector $[(q_0, i_0, \tau_0), \dots, (q_{k-1}, i_{k-1}, \tau_{k-1})] : \text{Vect}(\mathbb{S}\times\mathbb{N}\times\mathbb{D}^*)$ with $M(q_\ell) = (i_\ell, \tau_\ell)$, where the states are not duplicated and they are ordered by greedy priority: $(i_0, \tau_0) < (i_1, \tau_1) < \cdots < (i_{k-1}, \tau_{k-1})$. A triple $(q, i, \tau) : \mathbb{S} \times \mathbb{N} \times \mathbb{D}^*$ is called a *greedy token*.

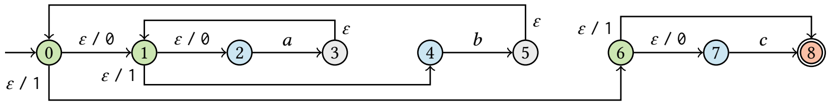
Let us discuss now some easy consequences of Def. 10. First, we observe that $\textsc{Paths}(w, i, i, q_{fin}) = \mathrm{P}_\varepsilon(q_{in}, q_{fin})$ for every i with $0 \le i \le |w|$. We have that $\text{GB}\textsc{est}(w) \simeq \text{GPr}(w[0..j], q_{fin})$ for some $j = 0, \dots, |w|$. From $\text{Acc}(w) \subseteq \text{Acc}(wa)$ we obtain that $\text{GB}\textsc{est}(wa) \le \text{GB}\textsc{est}(w)$. Finally, we observe that $\textsc{Matched}(w) = 1$ iff $\text{GB}\textsc{est}(w)$ is defined.

Lemma 11 (Greedy Trichotomy). Let $\mathcal{A}$ be a GNFA and $w \in \Sigma^*$. For every q that is guarded or final, $\text{GPr}(w, q) \ll \text{GB}\textsc{est}(w)$ or $\text{GPr}(w, q) \simeq \text{GB}\textsc{est}(w)$ or $\text{GB}\textsc{est}(w) \ll \text{GPr}(w, q)$.

Lemma 12 (Greedy Init & Step). Let $\mathcal{A}$ be a GNFA, $w \in \Sigma^*$, and $a \in \Sigma$. The following hold:

(1) If $q \in \textsc{Reach}(\varepsilon) = \mathrm{R}_\varepsilon(q_{in})$, then $\text{GPr}(\varepsilon, q) = (0, \tau)$ with $\tau = \mathrm{T}_\varepsilon(q_{in}, q)$.
(2) For every $q' \in \textsc{Reach}(w)$, we have that $\text{GPr}(wa, q') = (|wa|, \mathrm{T}_\varepsilon(q_{in}, q'))$ or $\text{GPr}(wa, q') = \text{GPr}(w, q) \cdot \mathrm{T}_\varepsilon(q + 1, q')$ for some guarded state q with $q \to^a q + 1$.
(3) If $\text{GB}\textsc{est}(wa) \neq \text{GB}\textsc{est}(w)$, then there is a guarded state q with $q \to^a q + 1$ such that $\text{GB}\textsc{est}(wa) = (i, |wa|, \tau_1\tau_2)$, where $(i, \tau_1) = \text{GPr}(w, q)$ and $\tau_2 = \mathrm{T}_\varepsilon(q + 1, q_{fin})$.

Example 13. The (streamlined) Thompson GNFA for the regex $(a^*b)^*c?$ is shown below:



The ε-reachable states from the initial state $q_{in} = 0$ are $\mathrm{R}_\varepsilon(q_{in}) = \{0, 1, 2, 4, 6, 7, 8\}$. Notice that:

$$\mathrm{T}_\varepsilon(0, 0) = \varepsilon \quad \mathrm{T}_\varepsilon(0, 1) = \mathtt{0} \quad \mathrm{T}_\varepsilon(0, 2) = \mathtt{00} \quad \mathrm{T}_\varepsilon(0, 4) = \mathtt{01} \quad \mathrm{T}_\varepsilon(0, 6) = \mathtt{1} \quad \mathrm{T}_\varepsilon(0, 7) = \mathtt{10}$$

and $\mathrm{T}_\varepsilon(0, 8) = \mathtt{11}$. So, $\text{GC}\textsc{fg}(\varepsilon) = [(2, 0, \mathtt{00}), (4, 0, \mathtt{01}), (7, 0, \mathtt{10})]$ and $\text{GB}\textsc{est}(\varepsilon) = (0, 0, \mathtt{11})$. We will consider now the successors of the guarded states 2, 4, 7 and their ε-closure.

$$2 \to^a 3 \quad \mathrm{R}_\varepsilon(3) = \{3, 1, 2, 4\} \qquad \mathrm{T}_\varepsilon(3, 3) = \varepsilon \quad \mathrm{T}_\varepsilon(3, 1) = \varepsilon \quad \mathrm{T}_\varepsilon(3, 2) = \mathtt{0} \quad \mathrm{T}_\varepsilon(3, 4) = \mathtt{1}$$
$$4 \to^b 5 \quad \mathrm{R}_\varepsilon(5) = \{5, 0, 1, 2, 4, 6, 7, 8\} \quad \mathrm{T}_\varepsilon(5, 5) = \varepsilon \quad \mathrm{T}_\varepsilon(5, q) = \mathrm{T}_\varepsilon(0, q) \text{ for } q \neq 5$$

We also have that $7 \to^c 8$ and $\mathrm{R}_\varepsilon(8) = \{8\}$. From $\text{GC}\textsc{fg}(\varepsilon)$ and $\text{GB}\textsc{est}(\varepsilon)$ we calculate $\text{GC}\textsc{fg}(a) = [(2, 0, \mathtt{000}), (4, 0, \mathtt{001})]$ and $\text{GB}\textsc{est}(a) = (0, 0, \mathtt{11})$. Informally, we can think that $\text{GC}\textsc{fg}(a)$ is obtained from the token $(2, 0, \mathtt{00})$ of $\text{GC}\textsc{fg}(\varepsilon)$ after taking the transition $2 \to^a 3$ and then following ε-paths. Notice that $(0, \mathtt{000}) = (0, \mathtt{00}) \cdot \mathrm{T}_\varepsilon(3, 2)$ and $(0, \mathtt{001}) = (0, \mathtt{00}) \cdot \mathrm{T}_\varepsilon(3, 4)$.

Lemma 14. Let $\mathcal{A}$ be the Thompson GNFA for the regex r and $w \in \Sigma^*$. If $[\![r]\!]_{\mathrm{G}}(w) = \text{None}$, then $\text{GB}\textsc{est}(w)$ is undefined. If $[\![r]\!]_{\mathrm{G}}(w) = \text{Some}([i, j])$, then $\text{GB}\textsc{est}(w) = (i, j, \tau)$ for some $\tau \in \mathbb{D}^*$.

A *(POSIX) priority* is a a natural number (which represents the start position of a path in a string). We use the order $\prec$ for POSIX priorities (elements of $\mathbb{N}$) and matches (elements of $\mathbb{N} \times \mathbb{N}$). We define $i \prec i'$ iff $i < i'$, $i \prec [i', j']$ iff $i < i'$, and $[i, j] \prec [i', j']$ iff $i < i'$ or ($i = i'$ and $j > j'$).

Definition 15 (POSIX Configuration). Let $\mathcal{A}$ be a GNFA. If $q \in \textsc{Reach}(w)$, we define the *POSIX priority* $\text{PPr}(w, q) = \min\{i \mid 0 \le i \le |w| \text{ and } \textsc{Paths}(w, i, |w|, q) \neq \emptyset\}$. The *best POSIX match* for w is $\text{PB}\textsc{est}(w) = \min\{[i, j] \mid (i, j, \pi) \in \text{Acc}(w)\}$. The *POSIX configuration* for w is a partial map $M = \text{PC}\textsc{fg}(w)$ whose domain contains those guarded states q of $\textsc{Reach}(w)$ that satisfy $\text{PPr}(w, q) \preceq \text{PB}\textsc{est}(w)$. Moreoever, $M(q) = \text{PPr}(w, q)$ for every $q \in \text{dom}(M)$.

We can represent $\textsc{PCfg}(w)$ as a vector $[(q_0, i_0), \ldots, (q_{k-1}, i_{k-1})] : \mathrm{Vect}(\mathbb{S} \times \mathbb{N})$ with $M(q_\ell) = i_\ell$, where the states are not duplicated and they are ordered by POSIX priority: $i_0 \leq i_1 \ldots \leq i_{k-1}$. A pair $(q, i) : \mathbb{S} \times \mathbb{N}$ is called a *POSIX token.*

Lemma 16 (POSIX Init & Step). Let $\mathcal{A}$ be a GNFA, $w \in \Sigma^*$, and $a \in \Sigma$. The following hold:

1. If $q \in \textsc{Reach}(\varepsilon) = \mathrm{R}_\varepsilon(q_{in})$, then $\mathrm{PPr}(\varepsilon, q) = 0$.
2. For every $q' \in \textsc{Reach}(w)$, we have that $\mathrm{PPr}(wa, q') = |wa|$ or $\mathrm{PPr}(wa, q') = \mathrm{PPr}(w, q)$ for some guarded state q with $q \to^a q + 1$ and $q' \in \mathrm{R}_\varepsilon(q + 1)$.
3. If $\textsc{PBest}(wa) \neq \textsc{PBest}(w)$, then there is a guarded state q with $q \to^a q + 1$ such that $\textsc{PBest}(wa) = [i, |wa|]$, where $i = \mathrm{PPr}(w, q)$ and $q_{\mathit{fin}} \in \mathrm{R}_\varepsilon(q + 1)$.

Lemma 17. Let $\mathcal{A}$ be the Thompson GNFA for the regular expression r and $w \in \Sigma^*$. If $[\![r]\!]_\mathrm{P}(w) = \mathrm{None}$, then $\textsc{PBest}(w)$ is undefined. If $[\![r]\!]_\mathrm{P}(w) = \mathrm{Some}([i, j])$, then $\textsc{PBest}(w) = [i, j]$.

Lemma 14 (resp., Lemma 17) says that the Thompson GNFA for r can be used to implement the greedy denotation $[\![r]\!]_\mathrm{G}$ (resp., the POSIX denotation $[\![r]\!]_\mathrm{P}$).

Greedy Execution. The main idea behind the greedy execution of a GNFA is to maintain a configuration as an ordered list $\mathrm{Vect}(\mathbb{S} \times \mathbb{N} \times \mathbb{D}^*)$ of tokens of the form (q, i, τ), which are ordered according to their greedy priority. The configuration includes only guarded states. Recall that the pair (i, τ) of the start position and the disambiguation trace is the greedy priority. The tokens are ordered first according to start position and then according to the lexicographic order on $\mathbb{D}^*$.

Since we are searching for a match that can start at any position in the input text, the automaton has to be "restarted" at every step, in order to consider every start position. This needs to happen until the first match is found (at some location $[i, j]$). After this point, we know that we should not continue "restarting" the automaton as it would consider possible matches of lower priority, which will never be part of the output. The GNFA execution proceeds from left to right, consuming one alphabet symbol at every step. We maintain the start and end position of the best match encountered so far. The configuration is always trimmed so that it only contains tokens that can potentially give a better match than the best one seen so far. At every step, there are two possibilities: (1) no new match is identified, or (2) a new match is identified, which is necessarily a strictly better match than the best one seen before (therefore, the "best match" has to be updated to the newly found one).

Example 18. Consider the regex r = `[bc]*c`. Here, σ stands for the character class $\{b, c\}$. The GNFA for $\sigma^* c$ has states $Q = \{0, 1, 2, 3, 4\}$ with $\Delta(0) = (1, 3)$, $\Delta(1) = \sigma$, $\Delta(2) = 0$, and $\Delta(3) = c$.

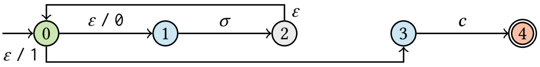


Table 1 shows the execution of the GNFA on the input string *abbcbcb*. For each prefix w of the input, the row "Greedy Configuration" shows $\textsc{GCfg}(w)$ and "Greedy Best Match" shows $\textsc{GBest}(w)$. We use the alternative notation $[i, j], \tau$ for a triple $(i, j, \tau) : \mathbb{N} \times \mathbb{N} \times \mathbb{D}^*$ representing a match.

POSIX Execution. In order to implement the POSIX semantics, the GNFA can be used but the disambiguation traces become irrelevant. This means that we can essentially view the GNFA as a classical ε-NFA. Recall that a *POSIX token* is of the form $(q, i) : \mathbb{S} \times \mathbb{N}$, where q is the state and i is the start position. As before, we record the start position because we are searching for a match that can occur anywhere in the input string. This means that the automaton has to be "restarted" at every step. For the POSIX semantics, priorities among tokens are determined purely by the start state: if $i < i'$, then the token (q, i) is of higher priority than (q, i'), because we prefer earlier matches. For this reason, the overall configuration can be represented as an element of $\mathrm{Vect}(\mathbb{S} \times \mathbb{N})$ or of

Table 1. Examples of GNFA Execution.

Computation of GNFA for the regex `[bc]*c`

Position	0	1	2	3	4	5	6	7
Input	a	b	b	c	b	c	b	
Greedy Configuration	(1, 0, 0) (3, 0, 1)	(1, 1, 0) (3, 1, 1)	(1, 1, 00) (3, 1, 01)	(1, 1, 000) (3, 1, 001)	(1, 1, 0000) (3, 1, 0001)	(1, 1, 00000) (3, 1, 00001)	(1, 1, 000000) (3, 1, 000001)	(1, 1, 0000000) (3, 1, 0000001)
Greedy Best Match	-	-	-	-	[1,4], 001	[1,4], 001	[1,6], 00001	[1,6], 00001

Computation of GNFA for the regex `a|ab`

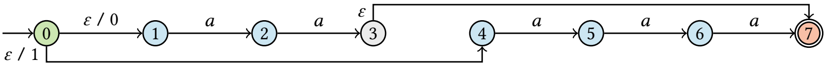
Position	0	1	2	3	4	5
Input	b	a	b	a	b	
G. Cfg	(1, 0, 0) (3, 0, 1)	(1, 1, 0) (3, 1, 1)	empty	empty	empty	empty
G. Match	-	-	[1,2],0	[1,2],0	[1,2],0	[1,2],0
P. Cfg	(1, 0) (3, 0)	(1, 1) (3, 1)	(4, 1)	empty	empty	empty
P. Match	-	-	[1,2]	[1,3]	[1,3]	[1,3]

Computation of GNFA for the regex `a*(ab)?`

Position	0	1	2	3
Input	a	a	b	
G. Cfg	(1, 0, 0) (4, 0, 10)	(1, 0, 00) (4, 0, 010)	(1, 0, 000) (4, 0, 0010)	empty
G. Match	[0,0], 11	[0,1], 011	[0,2], 0011	[0,2], 0011
P. Cfg	(1, 0) (4, 0)	(1, 0) (4, 0) (5, 0)	(1, 0) (4, 0) (5, 0)	empty
P. Match	[0,0]	[0,1]	[0,2]	[0,3]

Vect(Set($\mathbb{S} \times \mathbb{N}$)) by grouping tokens with the same start position. The latter is a representation where the tokens are *partially ordered* according to start position. An actual implementation could use the data structure Vect(Vect($\mathbb{S} \times \mathbb{N}$)) instead of Vect(Set($\mathbb{S} \times \mathbb{N}$)) without issue.

Example 19. To demonstrate how the partial order in POSIX, as opposed to the linear order in greedy, plays a role in the POSIX matching algorithm, consider the regular expression `aa|aaa` and input text *aaa*. The diagram below shows the GNFA for `aa|aaa`.



The table on the right shows the details of execution of the POSIX algorithm on the regular expression `aa|aaa` and text input *aaa*. At position 1, the tokens (2, 0) and (5, 0) were produced by moving forward the tokens from position 0, and the tokens (1, 1) and (4, 1) are added (i.e., "restart" the automaton to consider possible matches starting at index 1) because no match has been found yet. Observe that (2, 0) and (5, 0) are incomparable, and so are (1, 1) and (4, 1). However, both (2, 0) and (5, 0) are of higher priority than (1, 1) and (4, 1). Now, after moving forward the token (2, 0), POSIX reports a match [0, 2] at position and discards the lower-priority tokens (1, 1) and (4, 1). Note that due to the incomparability between (2, 0) and (5, 0), the token (5, 0) was not discarded but rather moved forward to (6, 0). By processing (6, 0) in position 2, POSIX reports the final best match [0, 3] in position 3.

Computation of GNFA for `aa|aaa`

Position	0	1	2	3
Input	a	a	a	
P. Cfg	(1, 0) (4, 0)	(2, 0) (5, 0) (1, 1) (4, 1)	(6, 0)	
P. Match	-	-	[0,2]	[0,3]

Example 20. The Thompson GNFA for the regular expression r = `a|ab` is the following:

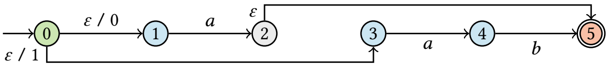


Table 1 shows the details of execution of the greedy and POSIX matching algorithms on the regular expression `a|ab` and text input *babab*. Note that the best matches eventually reported by the two algorithms differ. In particular, at position 1, both algorithms report the current best match [1,2]. However, the greedy algorithm returns early and disregards the token (3, 1, 1), which is of lower priority. The POSIX algorithm, on the other hand, moves forward the token $(3, 1)$ before returning and reporting the best match. Thus, at position 2, the greedy algorithm has run out of tokens, but the POSIX algorithm still has one more token to process, which in the end results in a longer match.

Example 21. The (streamlined) Thompson GNFA for the regex $r =$ `a*(ab)?` is the following:

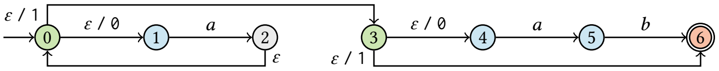


Table 1 shows the details of execution of the greedy and POSIX matching algorithms on the regular expression `a*(ab)?` and input text *aab*. Again, the behavior of the two algorithms diverges. The key observation here to understand the divergence is that, at position 1, the greedy algorithm processes the token $(1, 0, \texttt{00})$ and returns early to report the new best match $[0, 2], \texttt{0011}$ without processing the lower-priority token $(4, 0, \texttt{010})$ at all. This early return causes the greedy algorithm to miss one token at position 2 compared to the POSIX algorithm. By processing the token $(5, 0)$ at position 2, which the greedy algorithm misses, the POSIX algorithm finds the longest match $[0, 3]$. Although we see a difference in configuration at as early as position 1, the extra token that POSIX has there cannot be moved forward and thus has no impact on the divergence in the final output. It is the missing token at position 2 that prevents the greedy algorithm from finding the longest match.

4 STATIC ANALYSIS FOR ROBUSTNESS

In this section, we establish a characterization of non-robustness as a graph reachability property. The main idea is that we should consider greedy and POSIX configurations where the tokens are stripped down to contain only states (i.e., no start positions or disambiguation traces). This means that a greedy configuration is an ordered list of states, i.e., an element of $\mathrm{Vect}(Q)$. Similarly, a POSIX configuration is an ordered list of nonempty sets of states, i.e., an element of $\mathrm{Vect}(\mathrm{Set}(Q))$. Since no state repetition is allowed in configurations, there is an exponential number of them. We consider the product graph of greedy and POSIX configurations, which describes the parallel execution according to both greedy and POSIX semantics. Our characterization of non-robustness in terms of graph reachability (1) establishes that the problem of deciding robustness is in PSPACE and (2) provides an algorithm for checking robustness. We also provide two performance optimizations for the "base" reachability algorithm, which provide a substantial performance benefit according to our experimental evaluation of Section 5.

The algorithm for checking robustness is based on the idea that non-robustness can be identified by executing the GNFA (over all possible inputs) and identifying the case where the POSIX execution reports a new match (better than all previously identified matches) but the greedy execution does not report a new match (see Proposition 24). In order to do this, the GNFA execution does not need to record start positions and disambiguation traces in the tokens. It suffices to only keep an automaton state as a token, because we only need to indicate when a better match is found.

4.1 Stripped Greedy Execution

For the case of greedy GNFA execution, stripping the start positions and disambiguation traces from the token means that a configuration becomes a list of states (guarded states only, linearly ordered, no duplicate states), i.e., elements of $\mathrm{Vect}(Q)$, where $Q \subseteq \mathbb{S}$ is the set of states of the GNFA. The execution algorithm has to record whether a match has been found, because this affects whether the GNFA should be restarted or not. The configuration should only contain states that can identify a match that is strictly better than all the ones identified previously. This means that when a new match is found during the consumption of an input symbol, then the lower priority states that are awaiting processing should be discarded.

```
// Depth-first search (DFS) for greedy disambiguation
Function AddG($\mathcal{A}$ : GNFA($\Sigma$), $S$ : &mut Vect($\mathbb{S}$), $added$ : &mut Set($\mathbb{S}$), $q$ : $\mathbb{S}$):
    if $q \in added$ then return false
    $added$.insert($q$)   // insert $q$ into the set of “added” states
    if $\Delta(q) = \sigma \subseteq \Sigma$ then $S$.push($q$); return false
    else if $\Delta(q) = q' \in Q$ then return AddG($\mathcal{A}, S, added, q'$)
    else if $\Delta(q) = (q', q'') \in Q \times Q$ then
        $\mathbb{B}$ $b' \leftarrow$ AddG($\mathcal{A}, S, added, q'$); if $b'$ then return true // skip $q''$ (lower priority)
        return AddG($\mathcal{A}, S, added, q''$)
    else return true   // $q$ is the final state
// Returns the guarded states that are $\varepsilon$-reachable from the initial state.
Function InitialG($\mathcal{A}$ : GNFA($\Sigma$)):
    Vect($\mathbb{S}$) $S \leftarrow$ []; Set($\mathbb{S}$) $added \leftarrow \emptyset$   // no states have been added yet
    $\mathbb{B}$ $b \leftarrow$ AddG($\mathcal{A}$, &mut $S$, &mut $added$, 0)   // 0 is the initial state
    return $(S, b)$
// Returns the new configuration after consuming a symbol from the input.
Function NextG($\mathcal{A}$ : GNFA($\Sigma$), $matched$ : $\mathbb{B}$, $S$ : Vect($\mathbb{S}$), $a$ : $\Sigma$):
    Vect($\mathbb{S}$) $T \leftarrow$ []; Set($\mathbb{S}$) $added \leftarrow \emptyset$
    for $q = S[0], S[1], \ldots, S[S.\text{len} - 1]$ do // process tokens in order of priority
        if $a \in \Delta(q)$ then // $q$ is guarded, $\Delta$ is the transition function of $\mathcal{A}$
            if AddG($\mathcal{A}$, &mut $T$, &mut $added$, $q + 1$) then
                return ($T$, true)   // skip the rest of the states (lower priority)
    if $\neg matched$ then $\mathbb{B}$ $b \leftarrow$ AddG($\mathcal{A}$, &mut $T$, &mut $added$, 0) // should be $b$ = false
    return ($T$, false)
```

Fig. 2. Algorithm for identifying, at every step, whether a better greedy match is found.

The pseudocode of Fig. 2 gives the main ingredients of this process. The automaton $\mathcal{A}$ is meant to be the Thompson GNFA of the given regular expression r. The function `InitialG` computes the initial configuration, which is the set of guarded states that are ε-reachable from the initial state $q_{in} = 0$ (or, potentially, a subset of these if there is an ε-path from the initial state to the final state that is of higher priority than the ε-paths from the initial state to guarded states). The function `NextG` takes as input the current configuration S : Vect($\mathbb{S}$), a Boolean argument *matched* : $\mathbb{B}$ indicating whether a match has already been found, and the current input symbol $a : \Sigma$. It computes the next configuration by processing the tokens of S in order of priority, i.e., from higher priority to lower priority. A token $q : Q$ moves along an edge $q \rightarrow^{\sigma} q'$ with $q' = q + 1$ when $a \in \sigma$ and then all guarded states ε-reachable from q' are added to the next configuration. The processing of the tokens in S is terminated early if a match is found, in order to avoid paths of lower priority.

For a regular expression r, we define the ***Boolean greedy denotation*** $[r]_{\mathrm{G}} : \Sigma^* \rightarrow \mathbb{B}$ as follows:

$$[r]_{\mathrm{G}}(\varepsilon) = \begin{cases} 1, & \text{if } \mathcal{M}(\varepsilon, r) \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \qquad [r]_{\mathrm{G}}(wa) = \begin{cases} 1, & \text{if } [\![r]\!]_{\mathrm{G}}(wa) \neq [\![r]\!]_{\mathrm{G}}(w) \\ 0, & \text{otherwise} \end{cases}$$

Lemma 14 implies that the following are equivalent: $\mathcal{M}(w, r) \neq \emptyset$, $[\![r]\!]_{\mathrm{G}}(w) \neq$ None, $\textsc{Acc}(w) \neq \emptyset$, $\textsc{GBest}(w) \neq$ None, $\textsc{Matched}(w) = 1$. It follows that $\textsc{Matched}(w) = \bigvee_{i=0}^{|w|} [r]_{\mathrm{G}}(w[0..i])$. This is shown by induction on w (the Boolean denotation gives 1 when the first match appears).

Let S : Vect($\mathbb{S}$) and $w \in \Sigma^*$. For a GNFA $\mathcal{A}$, we write $S \approx \textsc{GCfg}(w)$ if S is the (unique) ***stripped representation*** of $\textsc{GCfg}(w)$, that is: (1) S has no duplicate states, (2) S contains exactly the states of the domain of $\textsc{GCfg}(w)$, and (3) S is ordered by the priorities specified by $\textsc{GCfg}(w)$.

Lemma 22. Let $\mathcal{A}$ be the Thompson GNFA for the regular expression r. The following hold:

(1) Let $(S, b) = \texttt{InitialG}(\mathcal{A})$. Then, $S \approx \textsc{GCfg}(\varepsilon)$ and $b = [r]_{\mathrm{G}}(\varepsilon)$.

(2) If $S \approx \textsc{GCfg}(w)$, then $S' \approx \textsc{GCfg}(wa)$ and $b = [r]_{\mathrm{G}}(wa)$, where (S', b) is the output of the function call $\texttt{NextG}(\mathcal{A}, \textsc{Matched}(w), S, a)$.

Proof. We will only consider Part (2) and focus on the case $\textsc{Matched}(w) = 1$. (The case $\textsc{Matched}(w) = 0$ is similar, with the only difference being that we need to also consider tokens with start position $|wa|$; the call to AddG in Line 20 adds those states.) The assumption $S \approx \textsc{GCfg}(w)$ says that S is linearly ordered by the greedy priority $\mathrm{GPr}(w, q)$ of the states q it contains and also that $\mathrm{GPr}(w, q) < \textsc{GBest}(w)$. The function NextG performs one computation step of $\mathcal{A}$. The crucial property is that every state of $\textsc{GCfg}(wa)$ can be reached from some state in $\textsc{GCfg}(w)$.

Let q' be a state in the domain of $\textsc{GCfg}(wa)$, i.e., q' is guarded and $\mathrm{GPr}(wa, q') < \textsc{GBest}(wa)$. Lemma 12 and $\textsc{Matched}(w) = 1$ imply that $\mathrm{GPr}(wa, q') = \mathrm{GPr}(w, q) \cdot \mathrm{T}_\varepsilon(q{+}1, q')$ for some guarded state q with $q \to^a q + 1$. It suffices to show that q is in $\textsc{GCfg}(w)$, i.e., $\mathrm{GPr}(w, q) < \textsc{GBest}(w)$. Lemma 11 says that (i) $\mathrm{GPr}(w, q) \ll \textsc{GBest}(w)$ or (ii) $\mathrm{GPr}(w, q) \simeq \textsc{GBest}(w)$ or (iii) $\textsc{GBest}(w) \ll \mathrm{GPr}(w, q)$. Case (ii) implies that $\mathrm{GPr}(wa, q') \geq \textsc{GBest}(w) \geq \textsc{GBest}(wa)$, which is a contradiction. Case (iii) implies that $\mathrm{GPr}(wa, q') \gg \textsc{GBest}(w) \geq \textsc{GBest}(wa)$, which is a contradiction. So, case (i) must hold, i.e., $\mathrm{GPr}(w, q) \ll \textsc{GBest}(w)$ and therefore $\mathrm{GPr}(w, q) < \textsc{GBest}(w)$. We have established that q is in $\textsc{GCfg}(w)$. This establishes that S contains all the states that are needed to obtain S'.

With similar arguments, we can also show that S contains all the states that are needed to identify when $\textsc{GBest}(wa) \neq \textsc{GBest}(w)$, which is equivalent to $[r]_{\mathrm{G}}(wa) = 1$.

Since NextG processes the states of S in order of priority, it is ensured that the states of S' appear in order of priority. Moreover, the exploration stops as soon as the final state is reached, which ensures that S' contains states with priority $< \textsc{GBest}(wa)$. Notice that finding a final state amounts to $\textsc{GBest}(wa) < \textsc{GBest}(w)$ and a return value $b = 1$. So, $b = [r]_{\mathrm{G}}(wa)$. □

4.2 Stripped POSIX Execution

For POSIX GNFA execution, stripping the start positions from the tokens means that a configuration becomes a list of nonempty sets of states (guarded states only, no duplicate states over the entire configuration), i.e., elements of $\mathrm{Vect}(\mathrm{Set}(Q))$, where $Q \subseteq \mathbb{S}$ is the set of states of the GNFA. So, a configuration has the form $[X_1, \ldots, X_k]$ where $\emptyset \neq X_i \subseteq Q$ for every i. The sets $X_1, \ldots, X_k$ are pairwise disjoint. For every i, the states within X_i are considered to be unordered (hence incomparable) because they have the same priority (the start position is the same for all of them). If $i < i'$, then the states in X_i are of higher priority than the states in $X_{i'}$. The main difference between the POSIX execution and the greedy execution has to do with the skipping of lower priority states. Notice in NextP of Fig. 3 that when a match is found we do not skip states from the same set X, only those from sets of lower priority (contrast this with NextG in Fig. 2). Also, notice in AddP of Fig. 3 that no state is skipped when finding the ε-reachable states because they are of the same priority (contrast this with AddG in Fig. 2). In contrast to the greedy case, a POSIX configuration is represented as a data structure of type $\mathrm{Vect}(\mathrm{Vect}(Q))$ (it is not an issue for the algorithm to represent $\mathrm{Set}(Q)$ as $\mathrm{Vect}(Q)$ because duplicate states are avoided over the entire configuration).

For a regular expression r, we define the ***Boolean POSIX denotation*** $[r]_{\mathrm{P}} : \Sigma^* \to \mathbb{B}$ as follows:

$$[r]_{\mathrm{P}}(\varepsilon) = \begin{cases} 1, & \text{if } \mathcal{M}(\varepsilon, r) \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \qquad [r]_{\mathrm{P}}(wa) = \begin{cases} 1, & \text{if } [\![r]\!]_{\mathrm{P}}(wa) \neq [\![r]\!]_{\mathrm{P}}(w) \\ 0, & \text{otherwise} \end{cases}$$

Lemma 17 implies that the following are equivalent: $\mathcal{M}(w, r) \neq \emptyset$, $[\![r]\!]_{\mathrm{P}}(w) \neq \mathrm{None}$, $\textsc{Acc}(w) \neq \emptyset$, $\textsc{PBest}(w) \neq \mathrm{None}$, $\textsc{Matched}(w) = 1$. It follows that $\textsc{Matched}(w) = \bigvee_{i=0}^{|w|} [r]_{\mathrm{P}}(w[0..i])$.

```
   // Depth-first search (DFS) for POSIX disambiguation
1  Function AddP($\mathcal{A}$ : GNFA($\Sigma$), $S$ : &mut Vect($\mathbb{S}$), $added$ : &mut Set($\mathbb{S}$), $q$ : $\mathbb{S}$):
2      if $q \in added$ then return false
3      $added$.insert($q$)   // insert $q$ into the set of “added” states
4      if $\Delta(q) = \sigma \subseteq \Sigma$ then $S$.push($q$); return false
5      else if $\Delta(q) = q' \in Q$ then return AddP($\mathcal{A}, S, added, q'$)
6      else if $\Delta(q) = (q', q'') \in Q \times Q$ then
7          $\mathbb{B}$ $b'$ ⟵ AddP($\mathcal{A}, S, added, q'$); $\mathbb{B}$ $b''$ ⟵ AddP($\mathcal{A}, S, added, q''$); return $b' \vee b''$
8      else return true   // $q$ is the final state
9  Function InitialP($\mathcal{A}$ : GNFA($\Sigma$)):
10     Vect($\mathbb{S}$) $S$ ⟵ []; Set($\mathbb{S}$) $added$ ⟵ $\emptyset$   // no states have been added yet
11     $\mathbb{B}$ $b$ ⟵ AddP($\mathcal{A}$, &mut $S$, &mut $added$, 0)   // 0 is the initial state
12     return ([$S$], $b$)
13 Function NextP($\mathcal{A}$ : GNFA($\Sigma$), $matched$ : $\mathbb{B}$, $S$ : Vect(Vect($\mathbb{S}$)), $a$ : $\Sigma$):
14     Vect(Vect($\mathbb{S}$)) $T$ ⟵ []; Set($\mathbb{S}$) $added$ ⟵ $\emptyset$
15     for $X = S[0], S[1], \ldots, S[S.\text{len} - 1]$ do // process tokens in order of priority
16         Vect($\mathbb{S}$) $Y$ ⟵ []; $\mathbb{B}$ $b$ ⟵ false   // $b$: match found within current priority?
17         for $q = X[0], X[1], \ldots, X[X.\text{len} - 1]$ do // all tokens in $X$ have the same priority
18             if $a \in \Delta(q)$ then // $\Delta$ is the transition function of $\mathcal{A}$
19                 if AddP($\mathcal{A}$, &mut $Y$, &mut $added$, $q + 1$) then $b$ ⟵ true
20         if $Y$.len > 0 then $T$.push($Y$)
21         if $b$ then return ($T$, true) // skip states of lower priority
22     if $\neg matched$ then
23         Vect($\mathbb{S}$) $Y$ ⟵ [];
24         $\mathbb{B}$ $b$ ⟵ AddP($\mathcal{A}$, &mut $Y$, &mut $added$, 0); assert $\neg b$;
25         if $Y$.len > 0 then $T$.push($Y$)
26     return ($T$, false)
```

Fig. 3. Algorithm for identifying, at every step, whether a better POSIX match is found.

Let S : Vect(Vect($\mathbb{S}$)) and $w \in \Sigma^*$. For a GNFA $\mathcal{A}$, we write $S \approx \text{PC}_{\text{FG}}(w)$ if S is a ***stripped representation*** of $\text{PC}_{\text{FG}}(w)$, that is: (1) S has no duplicate states, (2) S contains exactly the states of the domain of $\text{PC}_{\text{FG}}(w)$, (3) S is ordered by the priorities specified by $\text{PC}_{\text{FG}}(w)$, and (4) each inner vector contains all states of S with the same priority.

Lemma 23. Let $\mathcal{A}$ be the Thompson NFA for the regular expression r. The following hold:

(1) Let $(S, b) = \texttt{InitialP}(\mathcal{A})$. Then, $S \approx \text{PC}_{\text{FG}}(\varepsilon)$ and $b = [r]_{\text{P}}(\varepsilon)$.

(2) If $S \approx \text{PC}_{\text{FG}}(w)$, then $S' \approx \text{PC}_{\text{FG}}(wa)$ and $b = [r]_{\text{P}}(wa)$, where (S', b) is the output of the function call $\texttt{NextP}(\mathcal{A}, \text{Matched}(w), S, a)$.

Proof. For Part (1), notice that the domain of $\text{PC}_{\text{FG}}(\varepsilon)$ is $\text{Reach}(\varepsilon) = \text{R}_\varepsilon(q_{in})$, which is the set of all states that are ε-reachable from q_{in}. $\texttt{InitialP}(\mathcal{A})$ calls $\text{AddP}(\mathcal{A}, [\,], \emptyset, q_{in})$, which performs depth-first search to visit all states q that are ε-reachable from q_{in}. Out of these, only the guarded states are placed in the output vector S. The POSIX priority for every q in S is $\text{PPr}(\varepsilon, q) = 0$ by Lemma 16. $\texttt{InitialP}(\mathcal{A})$ returns $[S]$ for this reason. AddP returns $b = 1$ exactly when $q_{fin} \in \text{R}_\varepsilon(q_{in})$, which means that $b = [r]_{\text{P}}(\varepsilon)$. For Part (2), we will only consider the case $\text{Matched}(w) = 1$ (the case $\text{Matched}(w) = 0$ only differs in that states of priority $|wa|$ are added; see Lines 22–25 in the code). Suppose that $S \approx \text{PC}_{\text{FG}}(w)$. As in the greedy case, the key observation is that every state of $\text{PC}_{\text{FG}}(wa)$ can be reached from some state in S. Let q' be in the domain of $\text{PC}_{\text{FG}}(wa)$. That is, q' is guarded and $\text{PPr}(wa, q') \preceq \text{GBest}(wa)$. Lemma 16 implies that $\text{PPr}(wa, q') = \text{PPr}(w, q)$ for some guarded state q with $q \to^a q + 1$ and $q' \in \text{R}_\varepsilon(q + 1)$. It follows that $\text{PPr}(w, q) \preceq \text{GBest}(wa) \preceq$

GBest(w) and therefore q is in S. We have thus established that S contains all the states that are needed to compute S'. Using Lemma 16 (last part) and similar reasoning, we can also check whether PBest$(wa) \leq$ PBest(w) (which is equivalent to $[r]_{\mathrm{P}}(wa) = 1$) by finding a guarded state q in S with $q \to^{a} q+1$ and $q_{\mathit{fin}} \in \mathrm{R}_{\varepsilon}(q+1)$. When this happens, `NextP` skips all inner vectors (type $\mathrm{Vect}(\mathbb{S})$) of lower priority. So, $S' \approx$ PCfg(wa) and $b = [r]_{\mathrm{P}}(wa)$. □

4.3 Robustness Checking

Let r be a regular expression and $\mathcal{A} = (Q, \Delta)$ be its Thompson GNFA. The regex r is non-robust iff there exists some input string $w \in \Sigma^*$ for which the greedy and POSIX matching algorithms give different output. The only way that they can differ is with greedy returning some location $[i, j]$ and POSIX returning $[i, j']$ for some $j' > j$. This means that the non-robustness is witnessed by the prefix $w[0..j']$. As the greedy and POSIX execution perform a left-to-right pass over $w[0..j']$, the first disagreement occurs exactly at the end: POSIX reports a new best match, but greedy does not. This difference is based purely on reporting whether a new best match occurs or not, which is exactly what the algorithms of Fig. 2 and Fig. 3 do. Proposition 24 (below) formalizes this fact using the Boolean greedy and POSIX denotations from §4.1 and §4.2.

Proposition 24 (Robustness). For every regular expression r, $[\![r]\!]_{\mathrm{G}} = [\![r]\!]_{\mathrm{P}}$ iff $[r]_{\mathrm{G}} = [r]_{\mathrm{P}}$.

Proof. The left-to-right direction is immediate. For the right-to-left direction suppose that $[r]_{\mathrm{G}} = [r]_{\mathrm{P}}$. We will prove by induction that $[\![r]\!]_{\mathrm{G}}(w) = [\![r]\!]_{\mathrm{P}}(w)$ for every $w \in \Sigma^*$. First, we will deal with the base case $w = \varepsilon$. If $\mathcal{M}(\varepsilon, r) = \emptyset$, then $[\![r]\!]_{\mathrm{G}}(\varepsilon) = [\![r]\!]_{\mathrm{P}}(\varepsilon) = \mathrm{None}$. If $\mathcal{M}(\varepsilon, r) \neq \emptyset$, then $[\![r]\!]_{\mathrm{G}}(\varepsilon) = [\![r]\!]_{\mathrm{P}}(\varepsilon) = \mathrm{Some}([0,0])$. So, in all cases, $[\![r]\!]_{\mathrm{G}}(\varepsilon) = [\![r]\!]_{\mathrm{P}}(\varepsilon)$. For the step case, we consider a string wa, where $w \in \Sigma^*$ and $a \in \Sigma$. If $\mathcal{M}(wa, r) = \emptyset$, then $[\![r]\!]_{\mathrm{G}}(wa) = [\![r]\!]_{\mathrm{P}}(wa) = \mathrm{None}$. We assume from now on that $\mathcal{M}(wa, r) \neq \emptyset$. If $\mathcal{M}(w, r) = \emptyset$, then the new matches in $\mathcal{M}(wa, r)$ all have right endpoint $|wa|$. Since both Greedy and POSIX prefer the leftmost match, it follows that both $[\![r]\!]_{\mathrm{G}}(wa)$ and $[\![r]\!]_{\mathrm{P}}(wa)$ are equal to $\mathrm{Some}([i, |wa|])$ for some match $[i, |wa|] \in \mathcal{M}(wa, r)$ (i.e., i is the least start position among the new matches). From this point on, we consider the case $\mathcal{M}(w, r) \neq \emptyset$. The induction hypothesis gives us that $[\![r]\!]_{\mathrm{G}}(w) = [\![r]\!]_{\mathrm{P}}(w) = \mathrm{Some}([i, j])$ for some match $[i, j] \in \mathcal{M}(w, r)$. The new matches in $\mathcal{M}(wa, r) \setminus \mathcal{M}(w, r)$ all have right endpoint $|wa|$. We examine cases: (I) If there exists a new match $[i', |wa|]$ with $i' < i$ (choose the one with the least start position i'), then both Greedy and POSIX prefer this over $[i, j]$. So, $[\![r]\!]_{\mathrm{G}}(wa) = [\![r]\!]_{\mathrm{P}}(wa) = [i', |wa|]$. (II) Otherwise, if $[i, |wa|]$ is a new match, then POSIX prefers it over $[i, j]$ because it is longer. That is, $[\![r]\!]_{\mathrm{P}}(wa) = \mathrm{Some}([i, |wa|])$. This means that $[r]_{\mathrm{P}}(wa) = 1$ and hence $[r]_{\mathrm{G}}(wa) = 1$ from the assumption. It follows that Greedy has a new preferred match, which must be $[i, |wa|]$, since there are no other better matches. It follows that $[\![r]\!]_{\mathrm{G}}(wa) = \mathrm{Some}([i, |wa|]) = [\![r]\!]_{\mathrm{P}}(wa)$. (III) Otherwise, all the new matches are of the form $[i', |wa|]$ with $i' > i$, which means that $[\![r]\!]_{\mathrm{G}}(wa) = [\![r]\!]_{\mathrm{P}}(wa) = \mathrm{Some}([i, j])$. The proof is thus complete. □

Definition 25. Let $\mathcal{A}$ be a GNFA with states Q. We define the ***robustness graph*** of $\mathcal{A}$. This is a labeled transition system, denoted $\mathcal{R}(\mathcal{A})$ with vertices $St = St_G \times St_P \times \mathbb{B}$. The set $St_G \subseteq \mathrm{Vect}(Q)$ of stripped greedy configurations contains vectors $L = [q_0, \ldots, q_{k-1}]$ where each state of Q appears in L at most once. The set $St_P \subseteq \mathrm{Vect}(\mathrm{Vect}(Q))$ of stripped POSIX configurations contains lists of the form $L = [S_0, \ldots, S_{\ell-1}]$ where each S_i is nonempty and each state appears in L at most once. The third (Boolean) component of St is meant to record whether a match has been encountered so far. The initial vertex of $\mathcal{R}(\mathcal{A})$ is $\mathit{Init} = (S, T, b_1)$ with $(S, b_1) = \mathtt{InitialG}(\mathcal{A})$ and $(T, b_2) = \mathtt{InitialP}(\mathcal{A})$. For a vertex (S, T, m) and a letter $a \in \Sigma$, we define $\delta((S, T, m), a) = (S', T', m \vee b_1)$ and $\mathit{out}((S, T, m), a) = (b_1, b_2)$, where $(S', b_1) = \mathtt{NextG}(\mathcal{A}, m, S, a)$ and $(T', b_2) = \mathtt{NextP}(\mathcal{A}, m, T, a)$. We also define $\delta : \Sigma^* \to St$ by $\delta(\varepsilon) = \mathit{Init}$ and $\delta(wa) = \delta(\delta(w), a)$ for all $w \in \Sigma^*$ and $a \in \Sigma$.

```
// Check whether the input regular expression $r$ is disambiguation robust.
Function IsRobust($r : \mathrm{Reg}(\Sigma)$):
    $\mathcal{A} \leftarrow$ Thompson GNFA for $r$
    $(\mathrm{Vect}(\mathbb{S})\ S,\ \mathbb{B}\ b_1) \leftarrow$ InitialG($\mathcal{A}$)   // initial configuration for greedy
    $(\mathrm{Vect}(\mathrm{Vect}(\mathbb{S}))\ T,\ \mathbb{B}\ b_2) \leftarrow$ InitialP($\mathcal{A}$)   // initial configuration for POSIX
    assert $b_1 = b_2$   // Greedy and POSIX always agree on input $\varepsilon$
    $\mathbb{B}\ matched \leftarrow b_1$   // has a match been identified yet?
    $queue \leftarrow [(S, T, matched, \varepsilon)]$;  $set \leftarrow \{(S, T, matched)\}$   // data structures for BFS
    while $queue.\mathrm{len} > 0$ do // BFS over the product greedy/POSIX execution graph
        $(S, T, matched, w) \leftarrow queue.\mathrm{popFront}()$
        for $a \in \Sigma$ do
            $(S', b_1) \leftarrow$ NextG($\mathcal{A}, matched, S, a$)   // next configuration for greedy
            $(T', b_2) \leftarrow$ NextP($\mathcal{A}, matched, S, a$)   // next configuration for POSIX
            assert $\neg b_1 \vee b_2$ // $b_1 \Rightarrow b_2$: if greedy finds a new best match, then so should POSIX
            if $b_1 \neq b_2$ then return Some($wa$) // first disagreement between greedy and POSIX
            $matched \leftarrow matched \vee b_1$
            if $(S', T', matched) \notin set$ then $queue.\mathrm{pushBack}((S', T', matched, wa))$; $set.\mathrm{insert}((S', T', matched))$
    return None   // robust (no witness of non-robustness)
```

Fig. 4. Algorithm for checking whether a regex has the same semantics using the greedy and POSIX policies.

Lemma 26 (Non-robustness). A regular expression r is non-robust iff there exists $w \in \Sigma^*$ and $a \in \Sigma$ such that $out(\delta(w), a) = (b_1, b_2)$ with $b_1 \neq b_2$ in the robustness graph of $\mathcal{A}_r$.

Proof. We claim that $\delta(w) \approx (\textsc{GCfg}(w), \textsc{PCfg}(w), \textsc{Matched}(w))$ for every $w \in \Sigma^*$. This can be established with a straightforward induction, using Lemmas 22 and 23. From this claim and Lemmas 22 and 23 we obtain that $out(\delta(w), a) = ([r]_{\mathrm{G}}(wa), [r]_{\mathrm{P}}(wa))$ for every $w \in \Sigma^*$ and $a \in \Sigma$. The regex r is non-robust iff $[\![r]\!]_{\mathrm{G}} \neq [\![r]\!]_{\mathrm{P}}$, which is equivalent by Proposition 24 to $[r]_{\mathrm{G}}(w) \neq [r]_{\mathrm{P}}(w)$ for some $w \in \Sigma^*$. Since $[r]_{\mathrm{G}}(\varepsilon) = [r]_{\mathrm{P}}(\varepsilon)$, the witness of non-robustness must be some nonempty $wa \in \Sigma^+$. The result follows immediately. □

It follows from the previous discussion that finding a witness for the non-robustness of r can be done by exploring the vertices of the robustness graph $\mathcal{R}(\mathcal{A}_r)$ until a disagreement is found. There are polynomials $p(m)$ and $q(m)$ such that $|St_G| \leq 2^{p(m)}$ and $|St_P| \leq 2^{q(m)}$, where m is the size of r. So, the size of the robustness graph is $|St| \leq 2^{p(m)+q(m)+1}$.

Theorem 27 (Computational Complexity). The problem IsRobust is PSPACE-complete.

Proof. Theorem 5 establishes PSPACE-hardness. We write IsNonRobust for the complement of IsRobust. We describe a nondeterministic polynomial-space algorithm for IsNonRobust that searches for a witness of non-robustness in the graph $\mathcal{R}(\mathcal{A}_r)$ for a regex r: start with the initial vertex of $\mathcal{R}(\mathcal{A}_r)$ and then guess (and check) the path on some $w \in \Sigma^*$ (this is done by guessing w letter by letter, not all at once, and following the path) and $a \in \Sigma$ leading to a vertex $\delta(w)$ with $out(\delta(w), a) = (b_1, b_2)$ and $b_1 \neq b_2$. The correctness of the algorithm follows from Lemma 26. This nondeterministic algorithm needs a polynomial amount of space to store the current vertex of the robustness graph (which consists of a pair of configurations and a Boolean value). This establishes that IsNonRobust belongs to the complexity class NPSPACE, which is equal to PSPACE [Savitch 1970]. So, IsNonRobust is PSPACE-complete, and therefore IsRobust is also PSPACE-complete. □

Theorem 28 (Robustness Analysis). `IsRobust` of Fig. 4 solves the IsRobust problem.

Proof. The function IsRobust of Fig. 4 performs breadth-first search (BFS) in the robustness graph of $\mathcal{A}_r$ to find a non-robustness witness of minimum length. This approach is correct by Lemma 26. Every tuple (S, T, m, w) that is added to the search queue satisfies $(S, T, m) = \delta(w)$. The algorithm searches for a tuple (S, T, m, w) and a letter $a \in \Sigma$ satisfying $out((S, T, m), a) = (b_1, b_2)$ and $b_1 \neq b_2$. This is exactly the non-robustness criterion of Lemma 26. □

The algorithm IsRobust of Fig. 4 requires both exponential time and exponential space. Even though Savitch's algorithm could give us a polynomial-space algorithm for IsRobust, it would have much worse time complexity and would therefore be impractical. We use BFS as the graph exploration algorithm in order to construct witnesses of minimal length (easier to understand).

Example of Robustness Analysis. Consider the regex $r = a^*(ab)?$ over the alphabet $\Sigma = \{a, b\}$. The Thompson GNFA for r is shown in Example 21. The vertices of $\mathcal{R}(\mathcal{A}_r)$ explored by the static analysis algorithm of Fig. 4 correspond to pairs GCfg(w) and PCfg(w) for certain words $w \in \Sigma^*$.

These (stripped) configurations are shown in the table on the right. The algorithm starts from the empty string ε, and continues the graph exploration with the strings a, b, aa, and then ab. Since the string b produces empty configurations, the static-analysis algorithm does not need to check for extensions of b. After consuming the strings a and aa, both the greedy match and the POSIX match are updated, i.e, $[r]_G(a) = [r]_P(a) = [r]_G(aa) = [r]_P(aa) = 1$. Upon consuming the string b, neither the greedy nor the POSIX match are updated, i.e, $[r]_G(b) = [r]_P(b) = 0$. Thus, these are not witnesses of non-robustness. On the other hand, we have $[r]_G(ab) = 0$ and $[r]_P(ab) = 1$ since POSIX finds the new match $[0, 2]$ while the greedy match $[0, 1]$ remains the same. This makes the string ab a witness for non-robustness. The algorithm terminates when this witness is found and stops exploring the graph any further.

w	ε	a	b	aa	ab
GCfg$(w)\|_Q$	$[1, 4]$	$[1, 4]$	$[\,]$	$[1, 4]$	$[\,]$
$\min_< \mathcal{M}(w, r)$	[0, 0]	[0, 1]	[0, 0]	[0, 2]	[0, 1]
$[r]_G(w)$	1	1	0	1	0
PCfg$(w)\|_Q$	$[\{1, 4\}]$	$[\{1, 4, 5\}]$	$[\,]$	$[\{1, 4, 5\}]$	$[\,]$
$\min_< \mathcal{M}(w, r)$	[0, 0]	[0, 1]	[0, 0]	[0, 2]	[0, 2]
$[r]_P(w)$	1	1	0	1	1

4.4 Performance Optimizations

The hardness result of Theorem 5 suggests that there are regular expressions that are difficult to analyze. For this reason, we explore some optimizations that can speed up the algorithm of Fig. 4 for some classes of regular expressions.

Definition 29 (End-Ambiguity)**.** We say that a regular expression $r : \text{Reg}(\Sigma)$ (resp., automaton $\mathcal{A}$) is *end-ambiguous* if there is a string $u \in \Sigma^*$ and a nonempty string $v \in \Sigma^+$ such that $u, uv \in \mathcal{L}(r)$ (resp., $\mathcal{A}$ accepts both u and uv). We say that r is *end-unambiguous* if it is not end-ambiguous.

Proposition 30. If a regular expression is end-unambiguous, then it is also robust.

Proof. Suppose that $r : \text{Reg}(\Sigma)$ is end-unambiguous. Assume for contradiction that r is not robust. This means that there is a string $w \in \Sigma^*$ for which the greedy best match occurs at some location $[i, j]$ and the POSIX best match occurs at $[i, j']$ for some $j' > j$. Define $u = w[i..j]$ and $v = w[j..j']$. The string v is nonempty because $j' > j$. Moreover, $w[i..j] \in \mathcal{L}(r)$ and $uv = w[i..j'] \in \mathcal{L}(r)$, because they are matches. It follows that r is end-ambiguous, which is a contradiction. □

Given a regular expression r, checking end-ambiguity can be done in polynomial time. Let $\mathcal{A}$ be an NFA for r (it can be a Thompson or Glushkov NFA, it does not matter). Since r and $\mathcal{A}$ denote the same language, r is end-ambiguous iff $\mathcal{A}$ is end-ambiguous. We consider paths in the product automaton $\mathcal{A} \times \mathcal{A}$. Notice that $\mathcal{A}$ is end-ambiguous iff there exists a path

$$(q_0, q_0') \to^{a_1} (q_1, q_1') \to^{a_2} \cdots \to^{a_k} (q_k, q_k') \to^{b_1} (q_{k+1}, q_{k+1}') \to^{b_2} \cdots \to^{b_\ell} (q_{k+\ell}, q_{k+\ell}')$$

in $\mathcal{A} \times \mathcal{A}$ such that (1) q_0, q'_0 are initial states, (2) q_k is a final state, (3) $\ell \geq 1$, and (4) $q'_{k+\ell}$ is a final state. This can be checked with a quadratic-time algorithm $O(m^2)$, where m is the size of the regex.

Proposition 31 (Robustness Preservation)**.** Let r be a regular expression, σ be a character class, and $0 \leq m \leq n$ be integers. If r is robust, then so are $r\sigma$, $r\sigma^*$, $r\sigma?$, and $r\sigma\{m, n\}$.

Proof. First, we will establish a useful claim: A regular expression is not robust r iff there exists a string $w \in \Sigma^*$ such that $w \in \mathcal{L}(r)$ and the best greedy match of r in w is at some location $[0, j]$ with $j < |w|$. The right-to-left direction follows from the fact that the best POSIX match of r in w is at location $[0, |w|]$. For the left-to-right direction, assume that r is not robust. This means that there exists a string $u \in \Sigma^*$ such that the best greedy match of r in u is at some location $[i, j]$ and the best POSIX match is at $[i, j']$ for some $j' > j$. Define $w = u[i..j']$, which means that $w \in \mathcal{L}(r)$ (because it is a match). The best greedy match of r in w is at location $[0, j - i]$ and $j - i < |w| = j' - i$.

The proofs for all cases $r\sigma$, $r\sigma?$, $r\sigma^*$ and $r\sigma\{m, n\}$ rely on similar arguments. For this reason, we give the representative proof for $r\sigma^*$. Suppose that r is robust. Also, assume for contradiction that $r\sigma^*$ is not robust. It follows that there exists a string w with $w \in \mathcal{L}(r\sigma^*)$ such that the best greedy match is at some location $[0, k]$ with $k < |w|$. Let $[0, j][j, k]$ be the decomposition that witnesses this preferred greedy match, hence $w[0..j] \in \mathcal{L}(r)$ and $w[j..k] \in \mathcal{L}(\sigma^*)$. Let ℓ be the largest position such that $w[0..\ell] \in \mathcal{L}(r)$, which implies that and $w[\ell..|w|] \in \mathcal{L}(\sigma^*)$. Since $[0..j][j..k]$ is preferred (in the greedy semantics) over $[0..\ell][\ell..|w|]$, then $[0, j]$ is a preferred match for r than $[0..\ell]$. It cannot be that $j < \ell$, because then $[0, j]$ would be the preferred greedy match in $w[0..\ell]$, which contradicts the robustness of r. So, we know that $0 \leq \ell \leq j \leq k < |w|$. But $w[j..|w|] \in \mathcal{L}(\sigma^*)$ and the decomposition $[0, j][j, |w|]$ is preferred over $[0, j][j, k]$, which is a contradiction. □

Using the notion of end-ambiguity (see Definition 29 and Proposition 30) and the properties of robustness preservation (Proposition 31), we can describe three versions of our robustness analysis:

(1) **Version Base**: The algorithm `IsRobust` of Fig. 4 (without any further optimization).
(2) **Version Opt1**: First, check if the input regex r is end-unambiguous. If so, then r is declared to be robust. Otherwise, execute the algorithm `IsRobust` of Fig. 4.
(3) **Version Opt2**: This algorithm extends the version Opt1. First, check if the input regex r is end-unambiguous. If so, then r is robust. If r is end-ambiguous, then define r' to be the "right-trimmed" regular expression that results from r by removing trailing subexpressions of the form $\sigma, \sigma^*, \sigma?, \sigma\{m, n\}$. If r' is robust (check using `IsRobust`), then r is declared to be robust. Otherwise, execute `IsRobust` on r.

5 EXPERIMENTS

We have implemented the robustness algorithm presented in Section 4 using the Rust programming language. We perform an experimental evaluation to answer the following research questions:

(1) Does the issue of non-robustness arise in practice?
(2) Is our implementation practical for analyzing regular expressions that arise in real datasets?
(3) Do the optimizations of Section 4.4 provide a significant performance benefit?

There are many implementations of regex engines with a POSIX or greedy semantics. A list of several such engines is compiled in Table 2. Though most of those engines claim to either return the leftmost longest or the greedy match, there are some variations among them. For example, Berglund et al. [2021] have shown that the Boost semantics for capturing groups differs from POSIX. We will compare the POSIX and greedy semantics using the RE2 library [RE2 2024]. We have chosen RE2 because it supports both the POSIX and greedy semantics and it is a widely-used regex engine.

We use the following ***regex datasets*** that are derived from real applications: (1) the *Snort* [Roesch 1999; Snort 2024] and (2) *Suricata* benchmarks [Suricata 2024] that contain patterns for network

Table 2. Supported semantics for some widely-used regex engines

Engine	PCRE	GREP	TRE	Boost	C++	RE2	Rust	Python	Java	Go	Javascript	.NET
POSIX	No	Yes	Yes	Yes	No	Yes	No	No	No	Yes	No	No
Greedy	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

traffic, (3) the *SpamAssassin* benchmark [SpamAssassin 2024] for detecting spam email, and (4) the *RegexLib* benchmark [RegexLib 2024]. In total, we have collected more than 10,000 regexes.

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 and GCC/G++ 9.4.00. The RE2 library was installed from source available at [RE2 2024] with the 2023-11-01 release. For each experiment, we executed 10 trials and we report the mean.

Semantic Differences over Real Data. We have used two variants of RE2, which we call *RE2-Greedy* and *RE2-POSIX*. They only differ in the disambiguation policy that they use. We apply these engines to our datasets using real input text to discover output disagreements that are caused by the choice of disambiguation policy. This investigation has revealed that RE2-Greedy and RE2-POSIX produce different output on a large number of regexes in each dataset, and we present some examples below. The discovered output differences are evidence that *non-robustness* is a problem for regular expressions and input strings that arise in real applications.

Example 32. The regular expression `([a-z]{4,6})*([a-z]{2}==|[a-z]{3}=|[a-z]{4})` is from the Suricata dataset. The input text `\x0cmalwarebytes\x03org` is taken from a real PCAP network file. RE2-Greedy tries to match the character class `[a-z]` inside `[a-z]{4,6}` as many times as possible (i.e., 6 times). If `[a-z]{4,6}` is repeated twice to consume `malwarebytes`, then it is not possible to match the whole regex. So, RE2-Greedy repeats `[a-z]{4,6}` once over `malwar` and then uses `[a-z]{4}` from the nondeterministic choice over `ebyt`. This means that RE2-Greedy returns `malwarbyt` as the most preferred match. RE2-POSIX returns the leftmost longest match, which is `malwarebytes`.

Example 33. The regex `((\d|[1-9]\d)\.){3}(\d|[1-9]\d|1\d\d)` is from the Snort dataset. We will use the input text `utmb=64482928.4.8.1332657346264`, which is taken from a real PCAP file. Both RE2-Greedy and RE2-POSIX match the substring `28.4.8.` with the subexpression `((\d|[1-9]\d)\.){3}`. For the second subexpression, RE2-Greedy chooses `\d` and returns the overall match `28.4.8.1`. RE2-POSIX chooses `1\d\d` and returns the match `28.4.8.133`, which is longer.

Robustness Analysis over Real Datasets. To quantify how often the robustness issue can occur, we perform a robustness analysis using the algorithm presented in Section 4 over the datasets. This analysis shows that there are *hundreds* of non-robust regular expressions (>4% of all regexes) that could potentially lead to disagreements for some input. For each non-robust regex, we also produce a minimum length witness input text for which the POSIX and Greedy semantics disagree. Table 3 presents the performance results for the analysis using the 3 variants of the algorithm presented in Section 4: the Base algorithm presented in Fig. 4, and the two optimized versions Opt1 and Opt2 described in §4.4. Thanks to the analysis, we are able to identify which regexes are non-robust in only a few minutes. Due to a limit on the memory that can be used, a small number regexes cannot be analyzed. The non-robustness of these regexes can arise using very simple input strings, as we will show in the examples below taken

Table 3. Robustness Analysis: Total running time in minutes.

Dataset	Base	Opt1	Opt2
RegexLib	0.59	0.45	0.11
Snort	23.76	3.14	0.79
SpamAssassin	17.49	13.35	1.16
Suricata	23.08	2.85	0.48
All datasets	64.93	19.80	2.55

from the analysis results. Those results from Table 3 show that our static analysis can be used in practice over real datasets.

Example 34. The regex `(configdir|update|pluginmode)=.*(\|.+\||system)` from Snort is recognized as a non-robust regex by our static analysis with `update=\|system\|` as a minimal witness. The greedy policy chooses `update` for the first subexpression `(configdir|update|pluginmode)`, then it tries to consume as many characters as possible with `.*` but has to backtrack to match with the the second choice `system` from the subexpression `(\|.+\||system)`. So, the greedy output is `update=\|system`. The POSIX output is the entire witness `update=\|system\|` by matching ε with `.*` and taking the first choice `\| .+\|` from the subexpression `(\|.+\||system)`.

Example 35. The regex `([1-9][0-9]{0,7})+` from RegexLib is another example of a non-robust regular expression with `100000010` as a minimal witness. The greedy engine can only repeat `[1-9][0-9]{0,7}` once, because the inner bounded repetition `[0-9]{0,7}` consumes 7 characters (i.e., the maximum possible). So, the greedy match is `10000001`. The POSIX engine, on the other hand, produces the leftmost longest match `100000010` by repeating `[1-9][0-9]{0,7}` twice.

Effects of performance optimizations. Fig. 5 shows the performance of the basic version of the robustness algorithm (called base). The version that is called opt1 incorporates the optimization discussed in §4.4 (it checks for end-ambiguity first, which is much less costly than robustness, as checking for end-ambiguity can be done in polynomial time). The version opt2 builds upon opt1 by taking advantage of the preservation of robustness also presented in §4.4. Fig. 5 contains two rows of plots, one to compare base against opt1, and one to compare opt1 versus opt2. Each point in the plots corresponds to a regex. In total, the running time of the base algorithm presented in Table 3 is around 65 minutes for all datasets, and is further reduced by a factor of about 3× down to 20 minutes with opt1 and by a factor of around 25× thanks to opt2 down to a few minutes. Overall, the optimizations substantially reduce the running time, helping to reduce the number of regexes timed out to only a few with opt2. For the base algorithm, less than 3% of regexes cannot be analyzed, and this number goes down to 0.5% for opt1 and 0.04% for opt2. These results demonstrate the significant benefit of the optimizations, both in terms of running time and percentage of handled regexes.

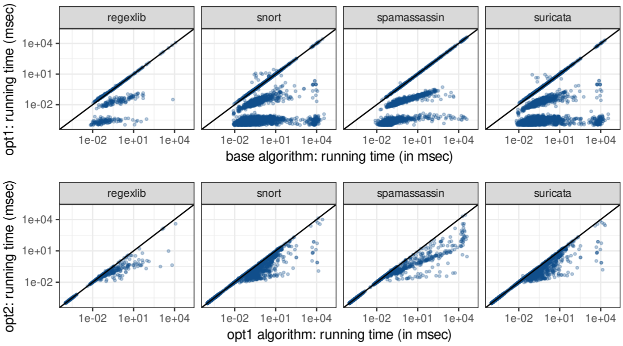


Fig. 5. Robustness Analysis: Comparison between the base algorithm and the optimized versions Opt1, Opt2.

6 RELATED WORK

Regex engines are either based on backtracking search (which may give rise to exponential running time) or automata (in which case they need linear time). The worst-case behavior of backtracking engines can be exploited to mount DoS attacks [Crosby and Wallach 2003; Davis et al. 2018; Staicu and Pradel 2018]. In spite of this, backtracking engines are still prevalent due to their support of extended features such as lookaround assertions and backreferences. Backtracking engines typically follow the greedy semantic of [PCRE 2024]. Frisch and Cardelli [2004] gave a formal exposition of this semantics by defining a linear order on parse trees. They also described a linear-time algorithm for constructing parse trees. Nielsen and Henglein [2011] use the term *bitcode* to describe the binary encoding of a parse tree. Grathwohl et al. [2013] propose an algorithm for greedy parsing that uses a smaller number of bits and Grathwohl et al. [2014] discuss a streaming version of the parsing problem. Cox [2010] has described a version of the problem where only the match interval (not the full parse tree) is of interest. The idea of tagging NFA transitions with priorities can be found in [Berglund and van der Merwe 2017; Laurikari 2000; Okui and Suzuki 2011].

The notion of ambiguity in NFAs [Book et al. 1971] and algorithms for computing its degree of growth have been studied in [Weber and Seidl 1991]. Ambiguity is relevant in the context of producing parse trees for matches. Kearns [1991] uses an automata-based algorithm to produce a linearized representation of a parse tree. Dubé and Feeley [2000] discuss how multiple parse trees can be implicitly encoded in a grammar. Sakuma et al. [2012] and Berglund and van der Merwe [2017] use transducers to produce parse trees. Borsotti et al. [2021]; Borsotti and Trofimovich [2021] consider NFA-based parsing and submatch extraction algorithms.

The choice of the parse tree is especially relevant in the presence of capture groups which extract the substring corresponding to the part of the subexpresion (sometimes refered to as *submatching*). Three ways of avoiding the problem of ambiguity are discussed in [Brabrand and Thomsen 2010]: rewriting ambiguous expressions away, introducing a restriction operator, and lazy/greedy annotations to each operator. In [Berglund and van der Merwe 2017], expressions with capture groups are formalized as transducers. They have extended their work [Berglund et al. 2017] to include *atomic* capture groups, a construct that prevents PCRE regexes from backtracking to retry matching. Laurikari [2000] invented the Tagged Determnistic Finite Automata (TDFA) to handle submatching. A version of this based on the POSIX policy was implemented by Chris Kuklewicz in 2007 and it was later improved by Borsotti and Trofimovich [2021]. The state-of-the-art implementation of TDFA-based algorithms is due to Trofimovich [2020].

Sulzmann and Lu [2012, 2014] use Brzozowski derivatives and partial derivatives for the purpose of parsing, including the handling of capture variables. They discuss how ambiguity can be detected in regular expressions using derivative-based techniques [Sulzmann and Lu 2016]. While the greedy semantics precisely specifies the most preferred parse tree, the disambiguation choices are less clear for POSIX sub-matching. Tan and Urban [2023]; Urban [2023] have formalized these details using Isabelle/HOL. Clarke and Cormack [1997] have considered shortest non-nested matches, motivated by the extraction of information from SGML documents. Yamamoto [2019] describe an efficient algorithm for finding all minimal matches. The semantics of capture groups in Boost regexes is considered by Berglund et al. [2021].

7 CONCLUSION

We have investigated the novel problem of deciding *disambiguation robustness*. Given a regular expression, this problem asks whether, for every input string, the match preferred by the greedy matching policy is the same as the one preferred by the POSIX policy. We have shown that this problem is PSPACE-complete and we have developed a static analysis algorithm for it. We have implemented the algorithm, as well as two performance optimizations. We thus provide the first tool that can identify non-robust regular expressions, which may be problematic for reuse in practice.

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 2313062.

REFERENCES

awk 2024. Gawk: Effective AWK Programming. https://www.gnu.org/software/gawk/. [Online; accessed March 24, 2024].

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.). Lecture Notes in Computer Science, Vol. 10457. Springer, Cham, 135–175. https://doi.org/10.1007/978-3-319-75632-5_5

Martin Berglund, Willem Bester, and Brink van der Merwe. 2021. Formalising and Implementing Boost POSIX Regular Expression Matching. *Theoretical Computer Science* 857 (2021), 147–165. https://doi.org/10.1016/j.tcs.2021.01.010

Martin Berglund and Brink van der Merwe. 2017. On the Semantics of Regular Expression Parsing in the Wild. *Theoretical Computer Science* 679 (2017), 69–82. https://doi.org/10.1016/j.tcs.2016.09.006

Martin Berglund, Brink van der Merwe, Bruce Watson, and Nicolaas Weideman. 2017. On the Semantics of Atomic Subgroups in Practical Regular Expressions. In *Implementation and Application of Automata (CIAA 2017) (Lecture Notes in Computer Science, Vol. 10329)*, Arnaud Carayol and Cyril Nicaud (Eds.). Springer, Cham, 14–26. https://doi.org/10.1007/978-3-319-60134-2_2

Ronald Book, Shimon Even, Sheila Greibach, and Gene Ott. 1971. Ambiguity in Graphs and Expressions. *IEEE Trans. Comput.* C-20, 2 (1971), 149–153. https://doi.org/10.1109/T-C.1971.223204

Angelo Borsotti, Luca Breveglieri, Stefano Crespi Reghizzi, and Angelo Morzenti. 2021. A Deterministic Parsing Algorithm for Ambiguous Regular Expressions. *Acta Informatica* 58, 3 (2021), 195–229. https://doi.org/10.1007/s00236-020-00366-7

Angelo Borsotti and Ulya Trofimovich. 2021. Efficient POSIX Submatch Extraction on Nondeterministic Finite Automata. *Software: Practice and Experience* 51, 2 (2021), 159–192. https://doi.org/10.1002/spe.2881

Claus Brabrand and Jakob G. Thomsen. 2010. Typed and Unambiguous Pattern Matching on Strings using Regular Expressions. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '10)*. ACM, New York, NY, USA, 243–254. https://doi.org/10.1145/1836089.1836120

Charles L. A. Clarke and Gordon V. Cormack. 1997. On the Use of Regular Expressions for Searching Text. *ACM Transactions on Programming Languages and Systems* 19, 3 (1997), 413–426. https://doi.org/10.1145/256167.256174

Russ Cox. 2010. Regular Expression Matching in the Wild. https://swtch.com/~rsc/regexp/regexp3.html. [Online; accessed March 24, 2024].

Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security Symposium*. USENIX Association, USA, 29–44. https://www.usenix.org/legacy/event/sec03/tech/full_papers/crosby/crosby.pdf

James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 246–256. https://doi.org/10.1145/3236024.3236027

James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-use and Portability of Regular Expressions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA, 443–454. https://doi.org/10.1145/3338906.3338909

Danny Dubé and Marc Feeley. 2000. Efficiently Building a Parse Tree from a Regular Expression. *Acta informatica* 37, 2 (01 Sep 2000), 121–144. https://doi.org/10.1007/s002360000037

Dominik D. Freydenberger. 2013. Extended Regular Expressions: Succinctness and Decidability. *Theory of Computing Systems* 53 (2013), 159–193. https://doi.org/10.1007/s00224-012-9389-0

Alain Frisch and Luca Cardelli. 2004. Greedy Regular Expression Matching. In *Automata, Languages and Programming (ICALP 2004) (Lecture Notes in Computer Science, Vol. 3142)*, Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.). Springer, Berlin, Heidelberg, 618–629. https://doi.org/10.1007/978-3-540-27836-8_53

Niels Bjørn Bugge Grathwohl, Fritz Henglein, Lasse Nielsen, and Ulrik Terp Rasmussen. 2013. Two-Pass Greedy Regular Expression Parsing. In *Implementation and Application of Automata (CIAA 2013) (Lecture Notes in Computer Science, Vol. 7982)*, Stavros Konstantinidis (Ed.). Springer, Berlin, Heidelberg, 60–71. https://doi.org/10.1007/978-3-642-39274-0_7

Niels Bjørn Bugge Grathwohl, Fritz Henglein, and Ulrik Terp Rasmussen. 2014. Optimally Streaming Greedy Regular Expression Parsing. In *Theoretical Aspects of Computing – ICTAC 2014 (Lecture Notes in Computer Science, Vol. 8687)*, Gabriel Ciobanu and Dominique Méry (Eds.). Springer, Cham, 224–240. https://doi.org/10.1007/978-3-319-10882-7_14

grep 2024. GNU Grep (Global Regular Expression Print). https://www.gnu.org/software/grep/. [Online; accessed March 24, 2024].

Renáta Hodován, Zoltán Herczeg, and Ákos Kiss. 2010. Regular Expressions on the Web. In *2010 12th IEEE International Symposium on Web Systems Evolution (WSE)*. IEEE, USA, 29–32. https://doi.org/10.1109/WSE.2010.5623572

Steven M. Kearns. 1991. Extending Regular Expressions with Context Operators and Parse Extraction. *Software: Practice and Experience* 21, 8 (1991), 787–804. https://doi.org/10.1002/spe.4380210803

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

Ville Laurikari. 2000. NFAs with Tagged Transitions, their Conversion to Deterministic Automata and Application to Regular Expressions. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval (SPIRE 2000)*. IEEE, USA, 181–187. https://doi.org/10.1109/SPIRE.2000.878194

Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2023. Regular Expression Matching using Bit Vector Automata. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1, Article 92 (2023), 30 pages. https://doi.org/10.1145/3586044

Konstantinos Mamouras and Agnishom Chattopadhyay. 2024. Efficient Matching of Regular Expressions with Lookaround Assertions. *Proceedings of the ACM on Programming Languages* 8, POPL, Article 92 (2024), 31 pages. https://doi.org/10.1145/3632934

Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes are Hard: Decision-making, Difficulties, and Risks in Programming Regular Expressions. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*. IEEE, USA, 415–426. https://doi.org/10.1109/ASE.2019.00047

Lasse Nielsen and Fritz Henglein. 2011. Bit-coded Regular Expression Parsing. In *Language and Automata Theory and Applications (LATA 2011) (Lecture Notes in Computer Science, Vol. 6638)*, Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide (Eds.). Springer, Berlin, Heidelberg, 402–413. https://doi.org/10.1007/978-3-642-21254-3_32

Satoshi Okui and Taro Suzuki. 2011. Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions. In *Implementation and Application of Automata (CIAA 2010) (Lecture Notes in Computer Science, Vol. 6482)*, Michael Domaratzki and Kai Salomaa (Eds.). Springer, Berlin, Heidelberg, 231–240. https://doi.org/10.1007/978-3-642-18098-9_25

PCRE. 2024. pcre2pattern man page. https://www.pcre.org/current/doc/html/pcre2pattern.html. [Online; accessed March 24, 2024].

RE2. 2024. RE2: Google's regular expression library. Website. https://github.com/google/re2 [Online; accessed March 24, 2024].

RegexLib. 2024. Regular Expression Library. https://regexlib.com/ [Online; accessed March 24, 2024].

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

Yuto Sakuma, Yasuhiko Minamide, and Andrei Voronkov. 2012. Translating Regular Expression Matching into Transducers. *Journal of Applied Logic* 10, 1 (2012), 32–51. https://doi.org/10.1016/j.jal.2011.11.003

Walter J. Savitch. 1970. Relationships Between Nondeterministic and Deterministic Tape Complexities. *J. Comput. System Sci.* 4, 2 (1970), 177–192. https://doi.org/10.1016/S0022-0000(70)80006-X

sed 2024. GNU sed (stream editor): non-interactive command-line text editor. https://www.gnu.org/software/sed/. [Online; accessed March 24, 2024].

Snort. 2024. Snort - Network Intrusion Detection & Prevention System. https://www.snort.org/ [Online; accessed March 24, 2024].

Apache SpamAssassin. 2024. Apache SpamAssassin. https://spamassassin.apache.org/ [Online; accessed March 24, 2024].

Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *27th USENIX Security Symposium (USENIX Security 2018)*. USENIX Association, USA, 361–376. https://www.usenix.org/conference/usenixsecurity18/presentation/staicu

Martin Sulzmann and Kenny Zhuo Ming Lu. 2012. Regular Expression Sub-Matching using Partial Derivatives. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming (PPDP '12)*. ACM, New York, NY, USA, 79–90. https://doi.org/10.1145/2370776.2370788

Martin Sulzmann and Kenny Zhuo Ming Lu. 2014. POSIX Regular Expression Parsing with Derivatives. In *Functional and Logic Programming (FLOPS 2014) (Lecture Notes in Computer Science, Vol. 8475)*, Michael Codish and Eijiro Sumii (Eds.). Springer, Cham, 203–220. https://doi.org/10.1007/978-3-319-07151-0_13

Martin Sulzmann and Kenny Zhuo Ming Lu. 2016. Derivative-Based Diagnosis of Regular Expression Ambiguity. In *Implementation and Application of Automata (CIAA 2016) (Lecture Notes in Computer Science, Vol. 9705)*, Yo-Sub Han and Kai Salomaa (Eds.). Springer, Cham, 260–272. https://doi.org/10.1007/978-3-319-40946-7_22

Suricata. 2024. Suricata - Open Source Intrusion Detection and Prevention Engine. https://suricata.io/ [Online; accessed March 24, 2024].

Chengsong Tan and Christian Urban. 2023. POSIX Lexing with Bitcoded Derivatives. In *14th International Conference on Interactive Theorem Proving (ITP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 268)*, Adam

Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 27:1–27:18. https://doi.org/10.4230/LIPIcs.ITP.2023.27

Ulya Trofimovich. 2020. RE2C: A Lexer Generator Based on Lookahead-TDFA. *Software Impacts* 6 (2020), 100027. https://doi.org/10.1016/j.simpa.2020.100027

Christian Urban. 2023. POSIX Lexing with Derivatives of Regular Expressions. *Journal of Automated Reasoning* 67, 3, Article 24 (2023), 24 pages. https://doi.org/10.1007/s10817-023-09667-1

Peipei Wang, Gina R. Bai, and Kathryn T. Stolee. 2019. Exploring Regular Expression Evolution. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, USA, 502–513. https://doi.org/10.1109/SANER.2019.8667972

Andreas Weber and Helmut Seidl. 1991. On the Degree of Ambiguity of Finite Automata. *Theoretical Computer Science* 88, 2 (1991), 325–349. https://doi.org/10.1016/0304-3975(91)90381-B

Ziyuan Wen, Lingkun Kong, Alexis Le Glaunec, Konstantinos Mamouras, and Kaiyuan Yang. 2024. BVAP: Energy and Memory Efficient Automata Processing for Regular Expressions with Bounded Repetitions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24)*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620665.3640412

Hiroaki Yamamoto. 2019. A Faster Algorithm for Finding Shortest Substring Matches of a Regular Expression. *Inform. Process. Lett.* 143 (2019), 56–60. https://doi.org/10.1016/j.ipl.2018.12.001

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 2023-11-16; accepted 2024-03-31