Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching

Martin Berglund
Department of Computing Science,
Umeå University,
Umeå, Sweden
mbe@cs.umu.se

Frank Drewes
Department of Computing Science,
Umeå University,
Umeå, Sweden
drewes@cs.umu.se

Brink van der Merwe
Department of Mathematical Sciences,
Computer Science Division,
University of Stellenbosch,
Stellenbosch, South Africa
abvdm@cs.sun.ac.za

We develop a formal perspective on how regular expression matching works in Java[1], a popular representative of the category of regex-directed matching engines. In particular, we define an automata model which captures all the aspects needed to study such matching engines in a formal way. Based on this, we propose two types of static analysis, which take a regular expression and tell whether there exists a family of strings which makes Java-style matching run in exponential time.

1 Introduction

Regular expressions constitute a concise, powerful, and useful pattern matching language for strings. They are commonly used to specify token lexemes for scanner generation during compiler construction, to validate input for web-based applications, to recognize meaningful patterns in natural language processing and data mining, for example, locating e-mail addresses, and to guard against computer system intrusion. Libraries for their use are found in most widely-used programming languages.

There are two fundamentally different types of regex matching engines: DFA (Deterministic Finite Automaton) and NFA (Non-deterministic Finite Automaton) matching engines. DFA matchers are used in (most versions of) awk, egrep, and in MySQL, and are based on the NFA to DFA subset conversion algorithm. This paper deals with NFA engines, which are found in GNU Emacs, Java, many command line tools, .NET, the PCRE (Perl compatible regular expressions) library, Perl, PHP, Python, Ruby and Vim. NFA matchers make use of an input-directed depth-first search on an NFA, and thus the matching performed by NFA engines is referred to as backtracking matching. NFA engines have made it possible to extend regular expressions with captures, possessive quantifiers, and backreferences.

Theory has however not kept pace with practice when it comes to understanding NFA engines. We now have NFA matchers that are more expressive and succinct than the originally developed DFA matchers, but are also in some cases significantly slower. Although it is known that in the worst case, the matching time of NFA matchers is exponential in the length of input strings [7], their performance characteristics and operational matching semantics are poorly understood in general. Exponential matching time, also referred to as catastrophic backtracking (by NFA matchers), can of course be avoided by using the DFA matchers, but then a less expressive pattern matching language has to be used. Catastrophic backtracking has potentially severe security implications, as denial-of-service attacks are possible in any application which matches a regular expression to data not carefully controlled by the application.

This work was motivated by the algorithm presented by Kirrage et. al. in [7], which for regular expressions with catastrophic backtracking comes up with a family of strings exhibiting this exponential

[1] Java is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Z. Ésik and Z. Fülöp (Eds.): Automata and Formal Languages 2014 (AFL 2014)
EPTCS 151, 2014, pp. 109–123, doi:10.4204/EPTCS.151.7
© M. Berglund, F. Drewes & B. v.d. Merwe

matching time behavior. However, they only consider the case where the exponential matching behavior can be exhibited by strings that are rejected. We investigate the complexity of deciding exponential backtracking matching on strings that are rejected (which we refer to as deciding exponential failure backtracking) further, and in addition we consider the general case of exponential backtracking. For this we introduce prioritized NFA (pNFA), which make non-deterministic choices in an ordered manner, thus prioritizing some over others in a way very reminiscent of parsing expression grammars (PEGs). The latter introduce ordered choice to the world of context-free grammars [5]. An interesting algorithm bridging the two areas is given in [8] by translating extended regular expressions to PEGs.

By linking failure backtracking with ambiguity in NFA, we show that catastrophic failure backtracking can be decided in polynomial time, and in the case of polynomial failure backtracking, the degree of the polynomial can be determined in polynomial time. General backtracking is shown decidable in EXPTIME by associating a tree transducer with the expression and applying a result from [4].

2 Preliminaries

For a set A, we denote by $\mathscr{P}(A)$ the power set of A. The constant function $f\colon A \to B$ with $f(a) = b \in B$ for all $a \in A$ is denoted by b^A. Also, given any function $f\colon A \to B$ and elements $a \in A$, $b \in B$, we let $f_{a\mapsto b}$ denote the function f' such that $f'(a) = b$ and $f'(x) = f(x)$ for all $x \in A \setminus \{a\}$. The set of all strings (or sequences) over A is denoted by A^*. In particular, it contains the empty string ε. To avoid confusion, it is assumed that $\varepsilon \notin A$. The length of a string w is denoted by $|w|$, and the number of occurrences of $a \in A$ in w is denoted by $|w|_a$. The union of disjoint sets A and B is denoted by $A \uplus B$.

As usual, a regular expression over an alphabet Σ (where $\varepsilon, \emptyset \notin \Sigma$) is either an element of $\Sigma \cup \{\varepsilon, \emptyset\}$ or an expression of one of the forms $(E \,|\, E')$, $(E \cdot E')$, or (E^*), where E and E' are regular expressions. Parentheses can be dropped using the rule that * (Kleene closure) takes precedence over $\cdot$ (concatenation), which takes precedence over $|$ (union). Moreover, outermost parentheses can be dropped, and $E \cdot E'$ can be written as EE'. The language $\mathscr{L}(E)$ denoted by a regular expression is obtained by evaluating E as usual, where $\emptyset$ stands for the empty language and $a \in \Sigma \cup \{\varepsilon\}$ for $\{a\}$.

A *tree* with labels in a set Σ is a function $t\colon V \to \Sigma$, where $V \subseteq \mathbb{N}_+^*$ is a non-empty, finite set of vertices (or nodes) which are such that (i) V is prefix-closed, i.e., for all $v \in \mathbb{N}_+^*$ and $i \in \mathbb{N}_+$, $vi \in V$ implies $v \in V$; and; (ii) V is closed to the left, i.e., for all $v \in \mathbb{N}_+^*$ and $i \in \mathbb{N}_+$, $v(i+1) \in V$ implies $vi \in V$.

The vertex ε is the root of the tree and vertex vi is the ith child of v. We let $|t| = |V|$ denote the size of t. t/v denotes the tree t' with vertex set $V' = \{w \in \mathbb{N}_+^* \mid vw \in V\}$, where $t'(w) = t(vw)$ for all $w \in V'$. If V is not explicitly named, we may denote it by $V(t)$. The *rank* of a tree t is the maximum number of children of vertices of t. Given trees $t_1, \ldots, t_n$ and a symbol α, we let $\alpha[t_1, \ldots, t_n]$ denote the tree t with $t(\varepsilon) = \alpha$ and $t/i = t_i$ for all $i \in \{1, \ldots, n\}$. The tree $\alpha[]$ may be abbreviated by α.

Given an alphabet Σ, the set of all trees of the form $t\colon V \to \Sigma$ is denoted by T_Σ. Moreover, if Q is an alphabet disjoint with Σ, we denote by $T_\Sigma(Q)$ the set of all trees $t\colon V \to \Sigma \cup Q$ such that only leaves may be labeled with symbols in Q, i.e., $t(v) \in Q$ implies that $v \cdot 1 \notin V$.

A *non-deterministic finite automaton* (NFA) is a tuple $A = (Q, \Sigma, q_0, \delta, F)$ where Q is a finite set of states, Σ is an alphabet with $\varepsilon \notin \Sigma$, $q_0 \in Q$, $F \subseteq Q$ and $\delta\colon Q \times (\{\varepsilon\} \cup \Sigma) \to \mathscr{P}(Q)$ is the transition function. The fact that $p \in \delta(q, \alpha)$ may also be denoted by $q \xrightarrow{\alpha} p$.

A *run* on a string $w \in \Sigma^*$ is a sequence $p_1 \cdots p_{m+1} \in Q^*$ such that there exist $\alpha_1, \ldots, \alpha_m \in \Sigma \cup \{\varepsilon\}$ with $\alpha_1 \cdots \alpha_m = w$ and $p_i \xrightarrow{\alpha_i} p_{i+1}$ for all $i \in \{1, \ldots, m\}$. Such a run is *accepting* if $p_1 = q_0$ and $p_m \in F$. The string w is accepted by A if and only if there exist an accepting run on w. The set of strings in Σ^* that are accepted by A is denoted by $\mathscr{L}(A)$.

A *string-to-tree transducer* is a tuple $stt = (Q, \Sigma, \Gamma, q_0, \delta)$, where Σ and Γ are the input and output alphabets respectively, Q is the finite set of states, $q_0 \in Q$ is the initial state, and $\delta\colon Q \times \Sigma \to T_\Gamma(Q)$ is the transition function. When $\delta(q, \alpha) = t$ we also write $q \xrightarrow{\alpha} t$.

For $\alpha_1, \ldots, \alpha_n \in \Sigma$, $stt(\alpha_1 \cdots \alpha_n)$ is the set of all trees $t \in T_\Gamma$ such that there exists a sequence of trees $t_0, \ldots, t_n$ which fulfill the requirement that $t_0 = q_0$ and $t_n = t$; and for every $i \in \{1, \ldots, n\}$, t_i is obtained from t_{i-1} by replacing every leaf v for which $t_{i-1}(v) \in Q$ with a tree in $\delta(t_{i-1}(v), \alpha_i)$, i.e., it holds that $t_i/v \in \delta(t_{i-1}(v), \alpha_i)$.

3 Regular Expression Matching in Java

Here we will take a look at the algorithm used for matching regular expressions in Java, using the default `java.util.regex` package, and describe in pseudocode how matching is accomplished in this package. The Java implementation is a good representative of the class of NFA search matchers. It is both fairly typical and very consistent across different versions (Java 1.6.0u27 is used to generate figures here). Many other implementations behave similarly, e.g. the popular Perl Compatible Regular Expressions library (PCRE). We try to capture the essence of the Java matching procedure as accurately as possible while omitting details, add-ons, and tricks that are irrelevant for the purpose of this paper.

Let us first describe the Java matcher in some detail. Readers who are not interested in this description may skip ahead to the second last paragraph before Algorithm 1. The core of the matcher is implemented in `java.lang.regex.Pattern`. Given a regular expression, it constructs an object graph of subclasses of the class `java.lang.regex.Pattern$Node` (we briefly call it `Node`, assuming all classes to be inner classes of `java.lang.regex.Pattern` unless otherwise stated). `Node` objects correspond to states, encapsulating their transitions in addition, and have one relevant method, `boolean Node.match(Matcher m, int i, CharSequence s)`, which we will closely mimic later. The implicit `this` pointer corresponds to the state, `s` is the entire string, `i` is the index of the next symbol to be read. The argument `m` contains a variety of book-keeping, notably variables corresponding to C in Algorithm 1 below, as well as after-the-fact information regarding the accepting run found. (In contrast, `match` returns `true` if and only if the node can, potentially recursively, match the remainder of the string). Every `Node` contains at least a pointer `next` which serves as the "default" next transition out of the node. Let us look at the object graph on the left in Figure 1. There are quite a few nodes even for a small expression like ab^*, but most are needed for fairly minor book-keeping, and for features we are not concerned with here. For example **LastNode** checks that all symbols are read by the matching, but can be made to do other things using additional features in `java.util.regex` which we do not deal with.

The matching starts with a call to `match` on **Begin** with the full string (i.e., `i` set to one and the string in `s`). See Figure 2 for pseudo-code for the behavior of **Begin**, **Single** and **Curly**. **Begin** (and **LastNode**) are trivial, they just check that we are in the expected position of the string, and in the case of **Begin** calls its `next`. **Single** reads a single symbol (equal to its internal variable `c`) and continues to `next`. **Accept** is even more trivial and always returns true. **Curly** handles the Kleene closure, and, since it has to resolve non-determinism (i.e. how many repetitions to perform), it is a bit more complex. The values `type`, `cmin`, and `cmax` are irrelevant for our concerns, they implement the counted repetition extension. Note that that line 2 in the right-most code snippet in Figure 2 works by updating values in the "`m`" in-out argument, but we leave that unspecific here. **Curly** starts by trying to match the atom node `atom` to a prefix of the string. If it succeeds it calls itself recursively, calling match on `this`, to process the remainder. When `atom` fails to match any further, **Curly** instead continues to `next` (backtracking as needed). In reality **Curly** uses imperative loops for efficiency, but it only serves to achieve a constant

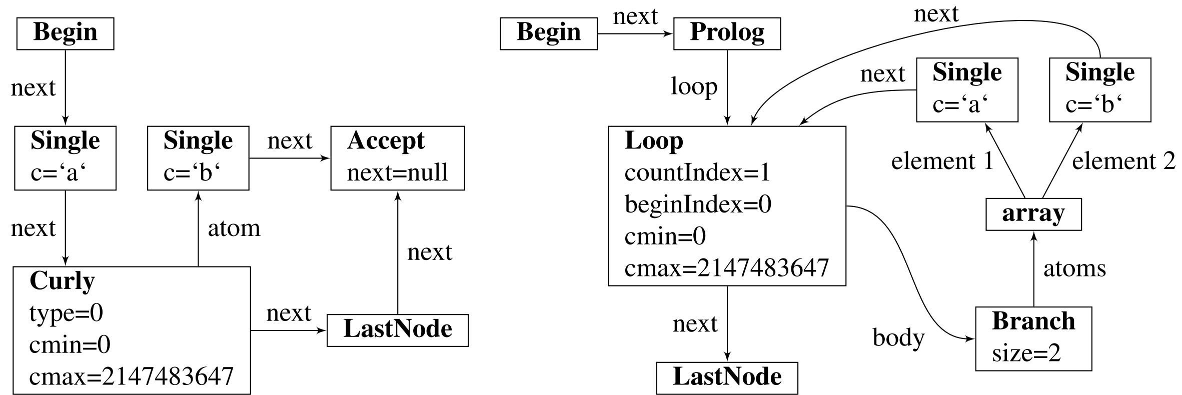


Figure 1: The left diagram shows essentially the complete internal object graph (i.e. internal data-structure) of subclasses of `Node` Java constructs for ab^*. On the right we show a simplified version of the corresponding object graph for $(a|b)^*$. In the latter all nodes without matching effect (in our limited expressions) are removed (e.g. the node **Accept** seen in the more complete example on the left).

```
1: if i = 0 then
2:     return next.match(i, w)
3: else
4:     return false
5: end if
```

```
1: if α_i = c then
2:     return next.match(i+1, w)
3: else
4:     return false
5: end if
```

```
1: if atom.match(i, w) then
2:     b := #symbols read above
3:     if this.match(i+b, w) then
4:         return true
5:     end if
6: end if
7: return next.match(i, w)
```

Figure 2: The code for a call of the form $\texttt{match}(i, w = \alpha_1 \cdots \alpha_n)$ on a **Begin** (left), **Single** (middle) and **Curly** (right) node. **Single** has a member variable `c` identifying the symbol it should read. **Curly** tries to recursively repeat `atom`, calling `next` when that fails.

speedup and is as such irrelevant for us. **Curly** is not used for all Kleene closures, if $b = 0$ it would loop forever, so the construction procedure for the object graph only uses **Curly** when it (with a fairly limited decision procedure) can tell that the contents looped is of constant non-zero length.

Next we look at the more general example on the right side of Figure 1. Here there are some additional nodes to consider. **Branch** implements the union, and **Prolog** and **Loop** implement the Kleene closure (with **Prolog** calling `matchInit` on **Loop** to initialize the loop). Let us look at each function in Figure 3. In **Branch**, `match` starts by letting the first subexpression match, continuing with the second and so on if the first attempts fail. The symbiotic relationship between **Prolog** and **Loop** is trickier. Where all other nodes calls into **Loop** with $\texttt{match}(i, w)$ as usual **Prolog** calls in with $\texttt{matchInit}(i, w)$ (on the left in Figure 3). This serves only one purpose: it eliminates ε-cycles. That is, it prevents **Loop** from recursively matching `body` to the empty string, making no progress. In `matchInit` the current value of i is stored, and in `match` (in the middle in Figure 3) an attempt to match `body` will only be made if at least one symbol has been read since the last attempt.

As an additional example, consider the regular expression $(a|a)^*$, which has an object graph almost like on the right of Figure 1, except the second **Single** *also* has `c` set to a. Matching this against $aa\cdots ab$ will take exponential time in the number of as, as all ways to match each a to each **Single** in $a|a$ will be tried as the matching backtracks trying to match the final b. In an experiment on one of the authors'

```
1: this.sp := i
2: if body.match(i, w) then
3:     return true
4: else
5:     return next.match(i, w)
6: end if
```

```
1: if i > this.sp then
2:     if body.match(i, w) then
3:         return true
4:     end if
5: end if
6: return next.match(i, w)
```

```
1: for e ∈ array do
2:     if e.match(i, w) then
3:         return true
4:     end if
5: end for
6: return false
```

Figure 3: The code for a call of the form $\texttt{match}(i, w = \alpha_1 \cdots \alpha_n)$ on a **Loop** (middle), **Branch** (right) and, as a special case, the call $\texttt{matchInit}(i, w)$ on **Loop** on the left. `matchInit` is called by **Prolog** in lieu of `match`. Notice that the loop in **Branch** is *in array order*.

desktop PCs an attempt to match $(a|a)^*$ to $a^{35}b$ using Java took roughly an hour of CPU time.

The object graph on the right in Figure 1 is, as is noted in the caption, a bit of creative editing of reality. A number of nodes not affecting the search behavior or matching are removed: the **Accept** node, which is just a `next` placeholder with no effect, **GroupHead** and **GroupTail**, which tracks what part of the match corresponds to a parenthesized subexpression, and finally **BranchConn**, which is placed in relation to **Branch** in the right of Figure 1 and records some information for the optimizer.

In general all nodes have numerous additional features not discussed, and there are many additional nodes serving similar purposes. For example **Single** may be replaced with **Slice** to match multiple symbols at once or **BnM** to matche multiple symbols using Boyer-Moore matching [2]. However, the optimizations are too minor to matter for our concerns (e.g. using **Slice** and **BnM** instead of **Single** yields at most a linear speed-up), and the additional features are outside our scope.

Let us take the above together and assemble the snippets of code into a function which takes a regular expression and a string as input and decides if the expression matches the string. A regular expression is represented by its parse tree, $T : \mathbb{N}_+^* \to \{|, {}^*, {}^{*?}, \cdot, \varepsilon\} \cup \Sigma$, defined in the obvious way with each $\cdot$ and $|$ having two children, * and $^{*?}$ one, and $\alpha \in \Sigma \cup \{\varepsilon\}$ zero. The operator $^{*?}$ is the lazy Kleene closure, which is the same as *, except that it attempts to make as *few* repetitions as possible.

We now define a function $\mathit{next} : \mathbb{N}_+^* \to \mathbb{N}_+^* \uplus \{\mathit{nil}\}$ on the nodes of T, where nil is a special value. Roughly speaking, $\mathit{next}(v)$ is the node at which parsing continues when the subexpression rooted av v has successfully matched (compare to the `cont` pointers in Kirrage at al. [7]). Let $\mathit{next}(\varepsilon) = \mathit{nil}$, and

1. if $T(v) = |$ then $\mathit{next}(v \cdot 1) = \mathit{next}(v \cdot 2) = \mathit{next}(v)$;
2. if $T(v) = \cdot$ then $\mathit{next}(v \cdot 1) = v \cdot 2$ and $\mathit{next}(v \cdot 2) = \mathit{next}(v)$; and
3. if $T(v) = {}^*$ or $T(v) = {}^{*?}$ then $\mathit{next}(v \cdot 1) = v$.

Then, collapsing the object graph and ignoring precise node choices in Java we get Algorithm 1.

Algorithm 1. *Simplified pseudocode of the Java matching algorithm. The implicit regular expression parse tree is T. The call-by-value input parameters are the node of T currently processed, the remainder of the input string, and a set of nodes that we should not revisit before consuming the next input symbol. This prevents ε-cycles as discussed above. The initial call made is* $\textsc{Match}(\varepsilon, w, \emptyset)$.

```
1:  function MATCH(v, w = a_1 ··· a_n, C)
2:      if v = nil then
3:          return n = 0
4:      else if T(v) = ε then
5:          return MATCH(next(v), w, C)
6:      else if T(v) ∈ Σ then
7:          if n ≥ 1 ∧ T(v) = a_1 then
8:              return MATCH(next(v), a_2 ··· a_n, ∅)
9:          end if
10:         return false
11:     else if T(v) = | then
12:         if MATCH(v·1, w, C) then
```

```
13:         return true
14:      end if
15:      return MATCH(v·2, w, C)
16:   else if T(v) = · then
17:      return MATCH(v·1, w, C)
18:   else if T(v) = * then
19:      if v·1 ∉ C then
20:         if MATCH(v·1, w, C ∪ {v·1}) then
21:            return true
22:         end if
23:      end if
24:      return MATCH(next(v), w, C)
25:   else if T(v) = *? then
26:      if MATCH(next(v), w, C) then
27:         return true
28:      else if v·1 ∉ C then
29:         return MATCH(v·1, w, C ∪ {v·1})
30:      else
31:         return false
32:      end if
33:   end if
34: end function
```

Notice how the code for the two Kleene closure variants *only* differs in what they try first: * tries to repeat its body first, whereas $^{*?}$ tries to not repeat the body. Note also how C is used to prevent ε-cycles in lines 19–20 and 28–29. If the node we *would* go to is already in C this means that no symbol has been read since last time we tried this, meaning repeating it would be a loop without progress.

4 Prioritized Non-Deterministic Finite Automata

We now define a modified type of NFA that provides us with an abstract view of the matching procedure discussed in the previous section. The modifications have no impact on the language accepted, but make the automaton "run deterministic". Every string in the language accepted has a unique accepting run, a property brought about by ordering the non-deterministic choices into a first, second, etc alternative, and letting the unique accepting run be given by trying, at any given state, alternative $i+1$ only when alternative i has failed. In our definition, only ε-transitions can be nondeterministic.

Definition 2. *A* prioritized non-deterministic finite automaton *(*pNFA*) is a tuple $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, where Q_1 and Q_2 are disjoint finite sets of states; Σ is a finite alphabet; $q_0 \in Q_1 \cup Q_2$ is the initial state; $\delta_1 \colon Q_1 \times \Sigma \to (Q_1 \cup Q_2)$ is the deterministic transition function; $\delta_2 \colon Q_2 \to (Q_1 \cup Q_2)^*$ is the non-deterministic prioritized transition function; and $F \subseteq Q_1 \cup Q_2$ are the final states.*

The NFA corresponding to the pNFA A is given by $\overline{A} = (Q_1 \cup Q_2, \Sigma, q_0, \bar{\delta}, F)$, where

$$\bar{\delta}(q,\alpha) = \begin{cases} \{\delta_1(q,\alpha)\} & \text{if } q \in Q_1 \text{ and } \alpha \in \Sigma, \\ \{q_1,\ldots,q_n\} & \text{if } q \in Q_2,\ \alpha = \varepsilon, \text{ and } \delta_2(q) = q_1 \cdots q_n. \end{cases}$$

The language accepted by *A, denoted by $\mathscr{L}(A)$, is $\mathscr{L}(\overline{A})$.*

Next, we define the so-called backtracking run of a pNFA on an input string w. This run takes the form of a tree which, intuitively, represents the attempts a matching algorithm such as Algorithm 1 would make until accepting the input string (or eventually rejecting it). The definition makes use of a parameter C whose purpose is to remember, for every state, the highest nondeterministic alternative that has been tried since the last symbol was consumed. This corresponds to the parameter C in Algorithm 1 and avoids infinite runs caused by ε-cycles.

Definition 3. *Let $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$ be a pNFA, $q \in Q_1 \cup Q_2$, $w = \alpha_1 \cdots \alpha_n \in \Sigma^*$, and $C \colon Q_2 \to \mathbb{N}$. Then the (q,w,C)*-backtracking run *of A is a tree over $Q_1 \cup Q_2 \uplus \{Acc, Rej\}$. It* succeeds *if and only if Acc occurs in it. We denote the (q,w,C)-backtracking run by $btr_A(q,w,C)$ and inductively define it as follows. If $q \in F$ and $w = \varepsilon$ then $btr_A(q,w,C) = q[Acc]$. Otherwise, we distinguish between two cases:*[2]

[2]For the first case, recall that 0^{Q_2} denotes the function $C \colon Q_2 \to \mathbb{N}$ such that $C(q) = 0$ for all $q \in Q_2$.

1. *If $q \in Q_1$, then*

$$btr_A(q,w,C) = \begin{cases} q[btr_A(\delta_1(q,\alpha_1),\alpha_2\cdots\alpha_n,0^{Q_2})] & \text{if } n>0 \text{ and } \delta_1(q,\alpha_1) \text{ is defined,} \\ q[Rej] & \text{otherwise.} \end{cases}$$

2. *If $q \in Q_2$ with $\delta_2(q) = q_1\cdots q_k$, let $i_0 = C(q)+1$ and $r_i = btr_A(q_i,w,C_{q\mapsto i})$ for $i_0 \le i \le k$. Then*

$$btr_A(q,w,C) = \begin{cases} q[Rej] & \text{if } i_0 > k, \\ q[r_{i_0},\ldots,r_k] & \text{if } i_0 \le k \text{ but no } r_i\ (i_0 \le i \le k) \text{ succeeds,} \\ q[r_{i_0},\ldots,r_i] & \text{if } i \in \{i_0,\ldots,k\} \text{ is the least index such that } r_i \text{ succeeds.} \end{cases}$$

The backtracking run of A on w *is* $btr_A(w) = btr_A(q_0,w,0^{Q_2})$*. If* $btr_A(w)$ *succeeds, then the* accepting run of A on w *is the sequence of states on the right-most path in* $btr_A(w)$*.*

Notice that the third parameter C in $btr_A(q,w,C)$ fulfills a similar purpose as the set C in Algorithm 1. It is used to track transitions that must not be revisited to avoid cycles.

Clearly, for a pNFA A and a string w, $w \in \mathscr{L}(A)$ if and only if $btr_A(w)$ succeeds, if and only if the accepting run of A on w is an accepting run of the NFA $\overline{A}$. Backtracking runs capture the behavior of the following algorithm which generalizes Algorithm 1 to arbitrary pNFAs to deterministically find the accepting run of A on w if it exists.

Algorithm 4. *Let $A = (Q_1,Q_2,\Sigma,q_0,\delta_1,\delta_2,F)$ be a pNFA. The call* $\textsc{Match}(q_0,w,0^{Q_2})$ *of the following procedure yields the accepting run of A on w if it exists, and $\perp \notin Q_1 \cup Q_2$ otherwise. The third parameter is similar to the C in Definition 3. For every state $q \in Q_2$ with out-degree d we have $C(q) \in \{0,\ldots,d\}$.*

```
 1: function MATCH(q, w = a_1 ··· a_n, C)
 2:   if q ∈ Q_1 then
 3:     if n = 0 then
 4:       if q ∈ F then
 5:         return q
 6:       else
 7:         return ⊥
 8:       end if
 9:     else
10:       return q · MATCH(δ_1(q, a_1), a_2 ··· a_n, 0^{Q_2})
11:     end if
12:   else
13:     if n = 0 ∧ q ∈ F then
14:       return q
15:     else
16:       q_1 ··· q_k := δ_2(q)
17:       for i = C(q)+1, ..., k do
18:         r := MATCH(q_i, w, C_{q↦i})
19:         if r ≠ ⊥ then
20:           return q · r
21:         end if
22:       end for
23:       return ⊥
24:     end if
25:   end if
26: end function
```

Notice especially line 10 where a symbol is read and C is reset to 0^{Q_2} in the recursive call. The case for Q_2 starts at line 13, the loop at 17 tries all not yet tried *transitions for that state. If no transition succeeds we fail on line 23.*

We note here that the running time of Algorithm 4 is exponential in general, just like Algorithm 1. This can be remedied by means of memoization, but potentially with a significant memory overhead, due to the fact that memoization needs to keep track of each possible assignment to all $C(q)$ with $q \in Q_2$.[3]

Depending on how one turns a given regular expression into a pNFA, Algorithm 4 will run more or less efficiently. For example, if the pNFA is built in a way that reflects Algorithm 1, analyzing the efficiency of Algorithm 4 or, equivalently, the size of backtracking runs, yields a (somewhat idealized) statement about the efficiency of the Java matcher.

[3] Apparently, starting from version 5.6, Perl uses memoization in its regular expression engine in order to speed up matching.

4.1 Two Constructions for Turning Regular Expressions into pNFA

In this section we give two examples of constructions that can be used to turn a regular expression E into a pNFA A such that $\mathscr{L}(A) = \mathscr{L}(E)$. The first is a prioritized version of the classical Thompson construction [9], whereas the second follows the Java approach.

Recall that the classical Thompson construction converts the parse tree T of a regular expression E to an NFA, which we denote by $Th(E)$, by doing a postorder traversal on T. An NFA is constructed for each subtree T' of T, equivalent to the regular expression represented by T'. We do not repeat this well-known construction here, assuming that the reader is familiar with it. Instead, we define a prioritized version, which constructs a pNFA denoted by $Th^p(E)$ such that $\overline{Th^p(E)} = Th(E)$.

Just as the construction for $Th(E)$, we define $Th^p(E)$ recursively on the parse tree for E. For each subexpression F of E, $Th^p(F)$ has a single initial state with no ingoing transitions, and a single final state with no outgoing transitions. The constructions of $Th^p(\emptyset)$, $Th^p(\varepsilon)$, $Th^p(a)$, and $Th^p(F_1 \cdot F_2)$, given that $Th^p(F_1)$ and $Th^p(F_2)$ are already constructed, are defined as for $Th(E)$, splitting the state set into Q_1 and Q_2 in the obvious way. It is only when we construct $Th^p(F_1|F_2)$ from $Th^p(F_1)$ and $Th^p(F_2)$, and $Th^p(F_1^*)$ from $Th(F_1)$, where the priorities of introduced ε-transitions require attention. We also consider the lazy Kleene closure $F_1^{*?}$, to illustrate the difference in priorities of transitions between constructions for the greedy and lazy Kleene closure. In each of the constructions below, we assume that $Th^p(F_i)$ ($i \in \{1,2\}$) has the initial state q_i and the final state f_i. Furthermore, δ_2 denotes the transition function for ε-transitions in the newly constructed pNFA $Th^p(E)$. All non-final states in $Th^p(E)$ that are in $Th^p(F_i)$ inherit their outgoing transitions from $Th^p(F_i)$.

- If $E = F_1|F_2$ then $Th^p(E)$ is built like $Th(E)$, thus introducing new initial and final states q_0 and f_0, respectively, and defining $\delta_2(q_0) = q_1q_2$ and $\delta_2(f_1) = \delta_2(f_2) = f_0$.
- If $E = F_1^*$ then we add new initial and final states q_0 and f_0 to Q_2 and define $\delta_2(q_0) = q_1f_0$ and $\delta_2(f_1) = q_1f_0$. The case $E = F_1^{*?}$ is the same, except that $\delta_2(q_0) = f_0q_1$ and $\delta_2(f_1) = f_0q_1$.

Thus, the pNFA $Th^p(F^*)$ tries F as often as possible whereas $Th^p(F^{*?})$ does the opposite.

The second pNFA construction is the one implicit in the Java approach and Algorithm 1. We denote this pNFA by $J^p(E)$. The base cases $J^p(\emptyset)$, $J^p(\varepsilon)$, $J^p(a)$ are identical to $Th^p(\emptyset)$, $Th^p(\varepsilon)$, $Th^p(a)$, respectively. Now, let us consider the remaining operators. Again, we assume that $J^p(F_i)$ ($i \in \{1,2\}$) has the initial state q_i and the final state f_i. Furthermore, δ_2 denotes the transition function for ε-transitions in the newly constructed pNFA $J^p(E)$.

- Assume that $E = F_1 \cdot F_2$. Then $J^p(E)$ is built from $J^p(F_1)$ and $J^p(F_2)$ by identifying f_1 with q_2, adding a new initial state $q_0 \in Q_2$ with $\delta_2(q_0) = q_1$, and making f_2 the final state. Thus, $J^p(E)$ is built like $Th^p(E)$, except that a new initial state is added and connected to the initial state of $J^p(F_1)$ by means of an ε-transition.
- If $E = F_1|F_2$ then $J^p(E)$ is constructed by introducing a new initial state q_0, defining $\delta_2(q_0) = q_1q_2$, and identifying f_1 and f_2, the result of which becomes the new final state.
- Now assume that $E = F_1^*$. Then we add a new final state f_0 to $J^p(F_1)$, make $q_0 = f_1$ the initial state of $J^P(E)$, and set $\delta_2(q_0) = q_1f_0$. The case $E = F_1^{*?}$ is exactly the same, except that $\delta_2(q_0) = f_0q_1$.

Observation 5. *Let E be a regular expression and A a pNFA. Then the running time of Algorithm 4 on w (with respect to E) is $\Theta(|btr_A(w)|)$.*

The two variants of implementing regular expressions by pNFA are closely related. In fact, Kirrage et al. [7] seem to regard them as being essentially identical and write that their reasons for choosing

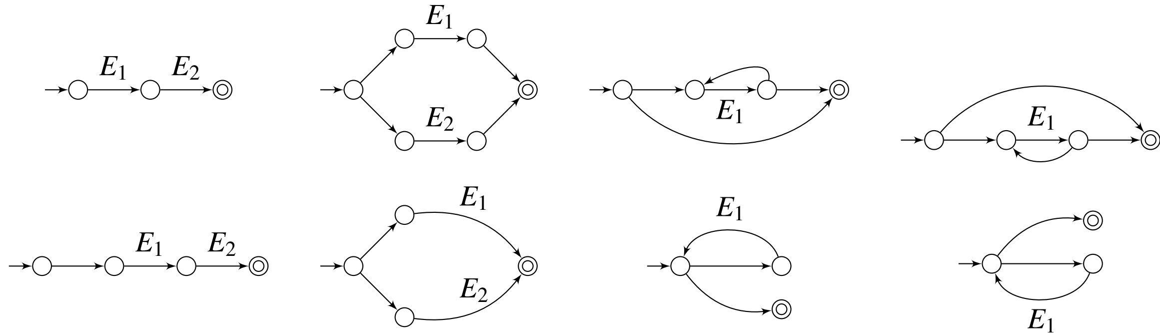
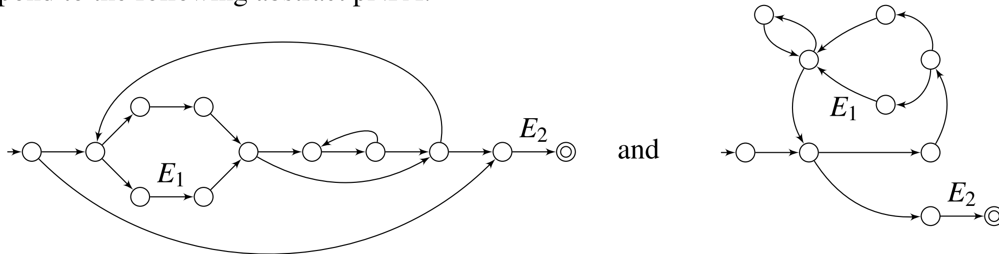


Figure 4: Abstract pNFA corresponding to $E_1 \cdot E_2$, $E_1 \,|\, E_2$, E_1^* and $E_1^{*?}$, from which $Th^p(E)$ (top row) and $J^p(E)$ (bottom row) are constructed. The transitions are prioritized in clockwise order, starting at noon.

$\overline{J^p(E)}$ are "purely of presentational nature". However, using our notion of pNFA we can show that this is not always the case. For this, note first that the construction of both $Th^p(E)$ and $J^p(E)$ can be viewed in a top-down fashion, where each operation is represented by an abstract pNFA in which zero, one, or two transitions are labeled with regular expressions. Replacing such a transition with the corresponding pNFA yields the constructed pNFA for the whole expression. Figure 4 shows the building blocks for the operations $\cdot$, $|$, *, and $^{*?}$ in both cases. Priorities follow the convention that ε-transitions leaving a state are drawn in clockwise order, starting at noon. Unlabeled edges denote ε-transitions.

Now consider an expression E of the form $((\varepsilon|E_1)\cdot\varepsilon^*)^*\cdot E_2$. When building $Th^p(E)$ and $J^p(E)$, these correspond to the following abstract pNFA:



In $Th^p(E)$, when processing an input string w, the run will first choose the prioritized choice of the union operator (which is ε), iterate the inner loop once, and then return to the initial state of the sub-pNFA corresponding to $\varepsilon|E_1$. Now, the first alternative is blocked, meaning that Algorithm 4 tries to match E_1. Assuming that no failure occurs, it will then proceed by following ε transitions leading to E_2.

Now look at $J^p(E)$. Here, the run first bypasses E_1, similarly to $Th^p(E)$, but this leads to the state following the start state. As the first alternative of transitions leaving this state has already been used, the run drops out of the loop and proceeds with E_2. E_1 will only be tried after backtracking in case E_2 fails.

We thus get several cases by appropriately instantiating E_1 and E_2. Assume first that we choose E_1 in such a way that $Th^p(E_1)$ suffers from exponential backtracking on a set W of input strings over Σ, and $E_2 = \Sigma^*$. Then $Th^p(E)$ causes exponential backtracking on strings in W whereas $J^p(E)$ does not backtrack at all. A concrete example is obtained by taking $\Sigma = \{a,b\}$, $E_1 = (a^*)^*$, and $W = \{a^n b \mid n \in \mathbb{N}\}$.

Conversely, we may choose $E_2 = \varepsilon \,|\, E_2'$ so that $J^p(E_2')$ fails exponentially on W, but $E_1 = \Sigma^*$. Then $Th^p(E)$ will match strings in W in linear time whereas $J^p(E)$ will take exponential time.

One can easily combine two examples of the types above into one, to obtain an expression such that $Th^p(E)$ shows exponential behavior on a set W of strings on which $J^p(E)$ runs in linear time whereas $J^p(E)$ shows exponential behavior on another set W' of strings on which $Th^p(E)$ runs in linear time.

5 Static Analysis of Exponential Backtracking

We now consider the problem of deciding whether a given pNFA causes backtracking matching similar to Algorithm 1 to run exponentially. More precisely, we ask whether a pNFA has exponentially large backtracking runs. In the case where the considered pNFA is $J^p(E)$, this yields a statement about the running time of Algorithm 1. However, we are interested in the problem in general, because other regular expression engines may correspond to other pNFA. There are two variants of the decision problem, with very different complexities. Let us start by defining the first.

Definition 6. *Given a pNFA* $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, *let* $f(n) = \max\{|btr_A(w)| \mid w \in \Sigma^*, |w| \leq n\}$ *for all* $n \in \mathbb{N}$. *We say that A has* exponential backtracking *if* $f \in 2^{\Omega(n)}$ *(or equivalently, if* $f(n) \in 2^{\Theta(n)}$*) and* polynomial backtracking of degree k *for* $k \in \mathbb{N}$ *if* $f \in \Theta(n^{k+1})$.

If the pNFA $A^f = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, \emptyset)$, *has exponential backtracking (or polynomial backtracking), then we say that A has* exponential failure backtracking *(polynomial failure backtracking, resp.).*

Failure backtracking provides an upper bound for the general case. In cases where the worst-case matching complexity can be exhibited by a family of strings not in $\mathscr{L}(A)$, this analysis is precise. This happens for example if for some $\$ \in \Sigma$, we have $w\$ \notin \mathscr{L}(A)$ for all $w \in \Sigma^*$, or more generally, if for each $w \in \Sigma^*$, there is $w' \in \Sigma^*$ such that $ww' \notin \mathscr{L}(A)$. Failure backtracking analysis is of great interest in that it is more efficiently decidable (being in PTIME) than the general case. It is closely related to the case considered in e.g. [7], where the matching complexity of the strings not in $\mathscr{L}(A)$ is studied.

5.1 An Upper Bound on the Complexity of General Backtracking Analysis

Let us first establish an upper bound on the complexity of general backtracking analysis. We will give an algorithm which solves this problem in EXPTIME. Afterwards, we will also note some minor hardness results. The EXPTIME decision procedure relies heavily on a result from [4].

Lemma 7. *Given a string-to-tree transducer* $stt = (Q, \Sigma, \Gamma, q_0, \delta)$, *it is decidable in deterministic exponential time whether the function* $f(n) = \max\{|t| \mid t \in stt(s),\ s \in \Sigma^*,\ |s| \leq n\}$ *grows exponentially, i.e. whether* $f \in 2^{\Omega(n)}$.

In short, we will hereafter construct a string-to-tree transducer from a pNFA A which reads an input string (suitably decorated) and outputs the corresponding backtracking run of A (see Definition 3). In this way, we model the running of Algorithm 4 on that string. Then Lemma 7 can be applied to this transducer to decide exponential backtracking. To simplify the construction we first make a small adjustment to the input pNFA in the form of a "flattening", which ensures that δ_2 maps Q_2 to Q_1^*. That is, we remove the opportunity for repeated ε-transitions.

Definition 8. *Let* $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$ *be a pNFA. Define* $d\colon (Q_1 \cup Q_2) \times (Q_2 \to \mathbb{N}) \to Q_1^*$, *and* $\bar{r}\colon Q_1^* \to Q_1^*$ *as follows:*

$$d(q, C) = \begin{cases} q & \text{if } q \in Q_1, \\ d(q_{i+1}, C_{q \mapsto i+1}) \cdots d(q_n, C_{q \mapsto i+1}) & \text{if } q \in Q_2,\ \delta_2(q) = (q_1 \cdots q_n) \text{ and } C(q) = i. \end{cases}$$

$$\bar{r}(s) = \begin{cases} \bar{r}(uv) & \text{if } s = uqv \text{ for some } u, v \in Q_1^* \text{ and } q \in Q_1 \text{ with } |u|_q \geq 2 \\ s & \text{otherwise.} \end{cases}$$

That is, $\bar{r}$ *removes all repetitions of each state q beyond the first two occurrences.*

Now, the δ_2-flattening *of A is the pNFA* $A' = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2', F')$ *with* $\delta_2'(q) = \bar{r}(d(q, 0^{Q_2}))$ *for all* $q \in Q_2$, *and* $F' = \{q \in Q_1 \cup Q_2 \mid d(q, 0^{Q_2}) \cap F \neq \emptyset\}$.

First let us note that the size of A' in Definition 8 is polynomial in the size of A, as no new states are added and no right-hand side is greater than polynomial in length ($2|Q_1|$ is the maximum length after applying $\bar{r}$). Furthermore, the construction itself can be performed in polynomial time in a straightforward way by computing d incrementally in a left-to-right fashion, and aborting each recursion visiting a state that has already been seen twice to the left.

Before proving some properties of the above construction we make a supporting observation.

Lemma 9. *Let σ be a function on trees such that, for $t = f[t_1,\dots,t_k]$*

$$\sigma(t) = \begin{cases} t & \text{if } k = 0 \\ f[\sigma(t_1)] & \text{if } k = 1 \\ f[\sigma(t_i),\sigma(t_j)] & \text{otherwise, where } t_i,t_j\ (i \neq j) \text{ are largest among } t_1,\dots,t_k. \end{cases}$$

Let $T_0,T_1,T_2,\dots$ be sets of trees of rank at most k. Then the function $f(n) = \max\{|t| \mid t \in T_n\}$ grows exponentially if and only if $f'(n) = \max\{|\sigma(t)| \mid t \in T_n\}$ grows exponentially.

We leave out the (rather easy) proof of the lemma due to space limitations.

Lemma 10. *Let $A = (Q_1,Q_2,\Sigma,q_0,\delta_1,\delta_2,F)$ be a pNFA and A' its δ_2-flattening. Then A' can be constructed in polynomial time, $\mathscr{L}(A') = \mathscr{L}(A)$, and the function $f(n) = \max\{|btr_A(w)| \mid w \in \Sigma^*,\ |w| \le n\}$ grows exponentially if and only if $f'(n) = \max\{|btr_{A'}(w)| \mid w \in \Sigma^*,\ |w| \le n\}$ grows exponentially.*

Proof sketch. Let $A' = (Q_1,Q_2,\Sigma,q_0,\delta_1,\delta_2',F')$. As noted, A' can be constructed in polynomial time.

The language equivalence of A and A' can be established by induction on the accepting runs of A and A'. δ_2' is a closure on δ_2, such that any accepting run for A of the form $p_1 \cdots p_n$ can be turned into one for A' by replacing each maximal subsequence $p_k \cdots p_{k+i} \in Q_2^*$ with just p_k. The function d in the construction of δ_2 will ensure that p_k is accepting if this was at the end of the run, and that p_k can go directly to the following Q_1 state. The converse is equally straightforward, as a suitable sequence from Q_2 can be inserted into an accepting run for A' to create a correct accepting run for A.

Finally, we argue that A' exhibits exponential backtracking behavior if and only if A does. By the construction of A', we have $btr_{A'}(w) \le btr_A(w)$. Hence, f grows exponentially if f' does. It remains to consider the other direction. Thus, assume that $f(n)$ grows exponentially. We have to show that $f'(n)$ grows exponentially as well. Let A'' be the pNFA generated by δ_2-flattening A without applying $\bar{r}$. Let $t = btr_A(w)$ and $t'' = btr_{A''}(w)$ for some input string w. Then t'' is obtained from t by repeatedly replacing subtrees of the form $q[s_1,\dots,s_k,q'[t_1,\dots,t_l],s_{k+1},\dots,s_m]$, where $q,q' \in Q_2$, by $q[s_1,\dots,s_k,t_1,\dots,t_l,s_{k+1},\dots,s_m]$. Since Definition 3 prevents repeated ε-cycles, this process removes only a constant fraction of the nodes in t.[4] Hence, $f''(n) = \max\{|btr_{A''}(w)| \mid w \in \Sigma^*,\ |w| \le n\}$ grows exponentially. Now, compare t'' with $t' = btr_{A'}(w)$. If a node of t'' has m children with the same state $q \in Q_2$ in their roots, by the definition of backtracking runs the m subtrees rooted at those nodes will be identical. This is the case since the run for each subtree starts in the same state and string position, and the application of d in the partial flattening ensures that C is made irrelevant by an immediately following δ_1 transition resetting it to 0^{Q_2}. The application of $\bar{r}$ to A'' means that, in effect, the first two copies of these m subtrees are kept in t'. In particular, the two largest subtrees of the node are kept in t'. According to Lemma 9, this means that g' grows exponentially. □

It should be noticed that, for the proof above to be valid, it is important that $\bar{r}$ preserves the order of occurrences of states from the left, as a subtree being accepting means that no further subtrees are constructed to the right of it (ensuring no extraneous subtrees get included).

[4] The constant may be exponential in the size of A, but for the question at hand this does not matter since the backtracking behavior in the length of the string is what is considered.

We are now prepared to define the construction which for any δ_2-flattened pNFA A produces a string-to-tree transducer *stt* such that $btr_A(w) = t$ if and only if $t \in stt(w')$. Here, w' is a version of w decorated with extra symbols $\flat$ and \$. The former will serve as padding to be read when δ_2 transitions are taken, and \$ marks the beginning and the end of the string.

Definition 11. *Given a δ_2-flattened pNFA $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$ we construct the string-to-tree transducer $stt = (Q, \Sigma', \Gamma, q'_0, \delta)$ in the following way. $Q = \{q'_0\} \cup \{a_q, f_q \mid q \in Q_1 \cup Q_2\}$, $\Sigma' = \Sigma \uplus \{\flat, \$\}$, and $\Gamma = Q_1 \cup Q_2 \uplus \{Acc, Rej\}$. Furthermore, δ consists of the following transitions:*

1. *Let $q'_0 \xrightarrow{\$} a_{q_0}$ and $q'_0 \xrightarrow{\$} f_{q_0}$. For all $q \in Q$ let $q \xrightarrow{\flat} q$.*
2. *For all $q \in Q_1$ and $\alpha \in \Sigma$:*
   (a) *If $\delta_1(q, \alpha) = q'$ let $a_q \xrightarrow{\alpha} q[a_{q'}]$ and $f_q \xrightarrow{\alpha} q[f_{q'}]$.*
   (b) *If $\delta_1(q, \alpha)$ is undefined let $f_q \xrightarrow{\alpha} q[Rej]$.*
3. *For all $q \in Q_2$, if $q_1 \cdots q_n = \delta_2(q)$, then for all $i \in \{0, \ldots, n-1\}$ let $a_q \xrightarrow{\flat} q[f_{q_1}, \ldots, f_{q_i}, a_{q_{i+1}}]$, and let $f_q \xrightarrow{\flat} q[f_{q_1}, \ldots, f_{q_n}]$.*
4. *Finally if $q \in F$ let $a_q \xrightarrow{\$} q[Acc]$, whereas when $q \notin F$:*
   (a) *if $q \in Q_1$ let $f_q \xrightarrow{\$} q[Rej]$, and,*
   (b) *if $q \in Q_2$ and $q_1 \cdots q_n = \delta_2(q)$, then $f_q \xrightarrow{\$} q[q_1[Rej], \ldots, q_n[Rej]]$.*

Definition 12. *The string $w_1\alpha_1 w_2\alpha_2 \cdots w_n\alpha_n w_{n+1}$ is a* decoration *of $\alpha_1 \cdots \alpha_n \in \Sigma^*$ if $w_i \in \{\$, \flat\}^*$ for each i. $\$\flat\alpha_1\flat\alpha_2 \cdots \flat\alpha_n\$$ is the* correct decoration *of $\alpha_1 \cdots \alpha_n$, denoted $dec(\alpha_1 \cdots \alpha_n)$.*

Lemma 13. *For a δ_2-flattened pNFA A, the string-to-tree transducer stt as constructed by Definition 11, and an input string $w = \alpha_1 \cdots \alpha_n$, it holds that $stt(dec(w)) = \{btr_A(w)\}$. For* all *$u$ which are decorations of w either $stt(u) = \emptyset$ or $stt(u) = \{btr_A(w)\}$.*

Proof sketch. First, notice how A being δ_2-flattened impacts btr_A. The flattening ensures that there is no way to take two ε-transitions in a row in A, meaning that every time case 2 of Definition 3 applies, we have $C(q) = 0$ since the previous step is either the initial call or a call from case 1 where C gets reset. As such we will have $C = 0^{Q_2}$ in every recursive call below. Let stt_q denote the string-to-tree transducer *stt* with the initial state q (instead of q_0).

Let $v = \$\flat\alpha_1\flat\alpha_2\flat \cdots \flat\alpha_n\$$. Establishing that $stt(dec(w)) = \{btr_A(w)\}$ merely requires a straightforward case analysis the details of which we leave out due to space limitations. Starting with the case where the backtracking run on w *fails*, the analysis establishes that for rejecting backtracking runs $t = btr_A(q, w, 0^{Q_2})$, we have $t \in stt_{f_q}(v)$, for all q, where v equals $dec(w)$ with the initial \$ removed (we will deal with this at the end) and, vice versa, $t \in stt_{f_q}(v)$ is true for exactly one t, so $t = btr_A(q, w, 0^{Q_2})$.

The proof for the accepting runs follows very similar lines, but with the extra wrinkle of how Q_2 rules are handled when some path accepts. The invariant that $t \in stt_{a_q}(v)$ is true for at most one t is maintained however, as is, of course, the parallel to btr_A. Again, the proof shows that $stt_{a_q}(v)$ outputs precisely one tree if v is $dec(w)$ with the initial \$ removed. That initial \$ is now used by the initial rules in *stt*: $q'_0 \xrightarrow{\$} a_{q_0}$ and $q'_0 \xrightarrow{\$} f_{q_0}$. This means that *stt* produces exactly one tree for every $dec(w)$, and in both the accepting and rejecting case it matches the tree from btr_A.

Finally, we need to deal with *incorrect* decorations. Let v be a decoration of w which is not $dec(w)$. If v has no leading \$, or no trailing \$, or has a \$ in any other position, $stt(v) = \emptyset$, since *stt* has no other possible rules for \$. If v contains *extraneous* $\flat$ we still have $stt(v) = \{btr_A(w)\}$, since they will just be consumed by $q \xrightarrow{\flat} q$ rules. If some $\flat$ is "missing" compared to $dec(w)$ this either causes $stt(v) = \emptyset$, if a Q_2 rule needed it, or $stt(v) = \{btr_A(w)\}$, if it is just removed by a $q \xrightarrow{\flat} q$ rule anyway. □

Theorem 14. *It is decidable in exponential time whether a given pNFA A has exponential backtracking.*

Proof. From A, construct the δ_2-flattened pNFA A' according to Definition 8. According to Lemma 10, A' can be constructed in polynomial time, and it has exponential backtracking if and only if A has. Construct the transducer *stt* for A' according to Definition 11. By Lemma 13 *stt* outputs exponentially large trees if and only if A' has exponential backtracking. The construction of *stt* can clearly be implemented to run in polynomial time. Hence, Lemma 7 yields the result. □

5.2 Hardness of General Backtracking Analysis

It seems likely that general backtracking analysis is computationally difficult. We cannot prove this yet, but here we demonstrate that either it is hard to decide if $J^p(E)$ has exponential backtracking *or* the class of regular expressions E such that $J^p(E)$ does *not* have exponential backtracking has an easy universality decision problem. In the following, we say that E has exponential backtracking if $J^p(E)$ does.

Let us briefly recall the universality problem.

Definition 15. *A regular expression E is* Σ-universal *if* $\Sigma^* \subseteq \mathscr{L}(E)$. *The input of* RE Universality *is an alphabet* Σ *and a regular expression E over* Σ. *The question asked is whether* $\mathscr{L}(E)$ *is* Σ*-universal.*

This problem is well-known to be PSPACE-complete. See e.g. [6]. We will now give a simple polynomial reduction which takes a regular expression E and constructs a new regular expression E' such that E' has exponential backtracking if E has exponential backtracking *or* E is not universal.

Lemma 16. *Let E be a regular expression over* Σ, $\alpha \in \Sigma$, *and* $\Gamma = \Sigma \cup \{\$\}$ *for some* $\$ \notin \Sigma$. *If E does not have exponential backtracking then* $E' = ((E \,|\, E\$\Gamma^*)\,|(\Sigma^*\$(\alpha^*)^*\$)$ *has exponential backtracking if and only if E is not* Σ*-universal.*

Proof. If E does not have exponential backtracking then neither does $E\$\Gamma^*$, since Γ^* never fails. Now, let $A = J^p(E')$. For every input string, the backtracking run of A will attempt to match $\Sigma^*\$(\alpha^*)^*\$$ to the string only if neither E nor $E\$\Gamma^*$ matches it. If E is universal, i.e. equal to Σ^*, then $\mathscr{L}(E|(E\$\Gamma^*)) = \mathscr{L}(\Sigma^*|(\Sigma^*\$\Gamma^*)) = \Gamma^*$ (since a string in Γ^* is either in Σ^* or has a prefix in Σ^* followed by a suffix in Γ^* that begins with a \$). Hence, in this case E' has exponential backtracking if and only if E does.

If we instead assume that E is *not* universal, then there exists some $w \in \Sigma^*$ such that $w \notin \mathscr{L}(E)$. Consider the string $w\$\alpha^n$ for any $n \in \mathbb{N}$. Neither E nor $E\$\Gamma^*$ matches it, which means that backtracking will proceed into $\Sigma^*\$(\alpha^*)^*\$$, where 2^n backtracking attempts will be made to match the suffix $\alpha^n\$$ to the subexpression $(\alpha^*)^*\$$ (as the final \$ keeps failing to match). □

The previous lemma yields the following corollary.

Corollary 17. *Let* $\mathscr{E}$ *be the set of all regular expressions that do* not *have exponential backtracking. Then either RE Universality is not PSPACE-hard for inputs in* $\mathscr{E}$, or *deciding whether regular expressions have exponential backtracking is PSPACE-hard.*

5.3 The Complexity of Failure Backtracking Analysis

Now we look at the problem to decide whether a given pNFA has exponential failure backtracking (see Definition 6). For reasons of technical simplicity, assume that parallel ε-transitions are absent from pNFA in this section. To simplify the exposition in this section, and to obtain a useful notion of ambiguity for NFA with ε-cycles, we restrict our notion of accepting runs of an NFA, as originally defined in Section 2. Consider a run $p_1 \cdots p_{m+1}$ on an input string $w = \beta_1 \cdots \beta_m \in \Sigma^*$. This run is called *short* if there are no

i, j, $1 \leq i < j \leq m$, such that $\beta_i = \ldots = \beta_j = \varepsilon$, $p_i = p_j$, and $p_{i+1} = p_{j+1}$. Thus, a short run must not contain any ε-cycle in which an ε-transition appears twice.

First we recall definitions from [1] on ambiguity for NFA, but for NFA with ε-cycles. These definitions differ from those in [1], due to the fact that we allow ε-cycles by using short accepting runs. We define the *degree of ambiguity* of a string w in N, denoted by $da(N,w)$, to be the number of short accepting runs in N labeled by w. N is *polynomially ambiguous* if there exists a polynomial h such that $da(N,w) \leq h(|w|)$ for all $w \in \Sigma^*$. The minimal degree of such a polynomial is the *degree of polynomial ambiguity* of N. We call N *exponentially ambiguous* if $g(n) = \max_{|w| \leq n} da(N,w) \in 2^{\Omega(n)}$ (or equivalently, if $g(n) \in 2^{\Theta(n)}$). It follows from Proposition 1 of [1] that N is either polynomially or exponentially ambiguous, i.e., there is nothing in between. To be precise, this concerns only NFA without ε-cycles, but as the proof of the following theorem shows, it extends to our more general case.

Theorem 18. *For an NFA N it is decidable in time $O(|N|_E^3)$ whether N is polynomially ambiguous, where $|N|_E$ denotes the number of transitions of N. If N is polynomially ambiguous, the degree of polynomial ambiguity can be computed in time $O(|N|_E^3)$.*

Proof. If N is ε-cycle free, the result follows from Theorems 5 and 6 in [1]. Now let $N = (Q, \Sigma, q_0, \delta, F)$ be an NFA, potentially with ε-cycles, and define the equivalence relation $\sim$ on Q, where $p \sim q$ if and only if they are in the same strongly connected component determined by using only ε-transitions in N. Let $N' := N/\sim$ be the quotient of N by $\sim$, having as states the equivalence classes of $\sim$.

The correctness of the remainder of the argument requires N not to have equivalence classes with two elements, say p, q, where both p and q do not have ε self-loops. We briefly argue how equivalences classes of this form can be removed without changing the ambiguity properties of N. It is tedious, but straightforward, to verify that this can for example be achieved by replacing p and q (and $p \xrightarrow{\varepsilon} q$, $q \xrightarrow{\varepsilon} p$) with 6 states and the appropriately defined ε-transitions to model the behavior of short runs in N that go through one or both consecutively of p and q. Three of the 6 states are used to model incoming transitions to p in (short) runs that after reaching p do not follow $p \xrightarrow{\varepsilon} q$, or follow only $p \xrightarrow{\varepsilon} q$, or follow consecutively $p \xrightarrow{\varepsilon} q$ and $q \xrightarrow{\varepsilon} p$, and the other 3 states are used for q in a similar way.

N' could potentially have (parallel) ε self-loops. Let N'' be N' with ε self-loops removed. Each state in N'' will belong to exactly one of the following categories of equivalence classes: (a) a single state of N without an ε self-loop in N; (b) a single state of N with an ε self-loop in N; (c) at least two states such that, in N, there are at least two distinct ε-runs (staying within the equivalence class) between any two states in the equivalence class (thanks to the modification of N described in the preceding paragraph).

Let Z be the set of states in N'' having the properties specified in (b) or (c). In N'' there are two possibilities. Either (i) there is a (short run which is a) cycle in N'' having at least one state in Z, or (ii) each short run in N'' goes through at most k states in Z (k is bounded by the number of states in N''). In case (i), N is exponentially ambiguous, since we have at least two ε-runs in N between any two states in an equivalence class in Z. In case (ii), the number of accepting runs in N'' (by definition without ε-cycles) and number of short accepting runs in N, differ by a constant factor, and we can apply the ε-cycle free result from [1] to N''. □

Theorem 19. *A pNFA A has either polynomial or exponential failure backtracking. It can be decided in time $O(|A|_E^3)$ whether A has polynomial failure backtracking, and if so, the degree of backtracking can be computed in time $O(|A|_E^3)$.*

Proof. Recall that A^f denotes the pNFA obtained from A where we change all states of A so that they are not accepting, and $\overline{A}^f$ denotes the NFA obtained by ignoring priorities on transitions of A^f. For an NFA

N, $a(N)$ is obtained from N by adding a new accepting sink state z (having transitions to itself on all input letters), all other states in N are made non-accepting, and we add ε-transitions from all states in N to z. Since $da(a(\overline{A}^f), w) = |btr_{A^f}(w)|$, and thus $\max\{da(a(\overline{A}^f), w) \mid w \in \Sigma^*, |w| \leq n\} = \max\{|btr_{A^f}(w)| \mid w \in \Sigma^*, |w| \leq n\}$, the failure backtracking complexity of A is equal to the ambiguity of $a(\overline{A}^f)$. To complete the proof, apply Theorem 18 to $a(\overline{A}^f)$. □

6 Conclusion/Future Work

Our prioritized NFA model is the only automata model, that we are aware of, which formalizes backtracking regular expression matching. This model is well suited to be extended to describe notions such as possessive quantifiers, captures and backreferences found in practical regular expressions. Backreferences have been formalized in [3], but without eliminating ambiguities due to multiple matches. Trying to improve our current complexity result for deciding backtracking complexity (as in Definition 6), and secondly, to formalize what is meant by equivalence of a regular expression with a pNFA, will provide the impetus for future investigations.

Acknowledgment We thank the referees for extensive lists of valuable comments.

References

[1] Cyril Allauzen, Mehryar Mohri & Ashish Rastogi (2008): *General Algorithms for Testing the Ambiguity of Finite Automata.* In Masami Ito & Masafumi Toyama, editors: *Developments in Language Theory*, *Lecture Notes in Computer Science* 5257, Springer, pp. 108–120, doi:10.1007/978-3-540-85780-8_8.

[2] Robert S. Boyer & J. Strother Moore (1977): *A fast string searching algorithm*. *Communications of the ACM* 20(10), pp. 762–772, doi:10.1145/359842.359859.

[3] Cezar Câmpeanu, Kai Salomaa & Sheng Yu (2003): *A Formal Study of Practical Regular Expressions*. *International Journal of Foundations of Computer Science* 14(6), pp. 1007–1018, doi:10.1142/S012905410300214X.

[4] Frank Drewes (2001): *The Complexity of the Exponential Output Size Problem for Top-Down and Bottom-Up Tree Transducers,*. *Information and Computation* 169(2), pp. 264 – 283, doi:10.1006/inco.2001.3039.

[5] Bryan Ford (2004): *Parsing Expression Grammars: A Recognition-Based Syntactic Foundation*. In Neil D. Jones & Xavier Leroy, editors: *Symposium on Principles of Programming Languages (POPL'04)*, ACM Press, pp. 111–122, doi:10.1145/964001.964011.

[6] Michael R. Garey & David S. Johnson (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

[7] James Kirrage, Asiri Rathnayake & Hayo Thielecke (2013): *Static Analysis for Regular Expression Denial-of-Service Attacks*. In: *Network and System Security*, Springer, pp. 135–148, doi:10.1007/978-3-642-38631-2_11.

[8] Sérgio Medeiros, Fabio Mascarenhas & Roberto Ierusalimschy (2012): *From Regexes to Parsing Expression Grammars*. *CoRR* abs/1210.4992, doi:10.1016/j.scico.2012.11.006.

[9] Ken Thompson (1968): *Regular Expression Search Algorithm.* *Communications of the ACM* 11(6), pp. 419–422, doi:10.1145/363347.363387.