Abstract

Regular expressions (*regex*) are a foundational tool in modern software, widely used for string matching, input validation, and text parsing. Despite their utility, improperly constructed regular expressions can introduce serious security vulnerabilities. One of the most critical threats is the Regular Expression Denial of Service (*ReDoS*) attack, in which carefully crafted inputs cause the regex engine to perform excessive and redundant processing. This results in dramatic slowdowns or even complete unresponsiveness of the system. ReDoS poses a significant risk to web applications, APIs, and other input-facing systems, where user-controlled input is matched against vulnerable patterns.

In this work, we propose a system to address ReDoS by transforming regular expressions into a modified *position automata*, a form of nondeterministic finite automaton (NFA) that tracks the exact start and end positions of all matches within an input string. This structure enables a matching function that computes *all* match positions, including overlapping ones, without relying on backtracking. By exhaustively and efficiently exploring the automaton's transitions, our approach avoids the exponential blowup typical of vulnerable engines, while preserving a somewhat full regex expressiveness.

Furthermore, we also review and compare this approach with existing solutions present in state-of-the-art programming languages and libraries, such as *RE#*. **Palavras-chave:** regular expressions, ReDoS, position automata, nondeterministic finite automata, pattern matching.

Resumo

O teu resumo COOL, its me TEST WOWIEESSS

Palavras-chave: palavra, chave..

Acknowledgements

First of all, I would like to thank my family, etc, etc

Dedico a . . . mim

Contents

List of Tables

List of Figures

Listings

Acronyms

BS	Base Station
BSN	Body Sensor Network
DCC	Departamento de Ciência de Computadores
ECG	ElectroCardioGram
FCUP	Faculdade de Ciências da Universidade do Porto
HR	Heart Rate
HTTP	Hypertext Transfer Protocol
MRI	Magnetic Resonance Imaging
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
USF	Unidade de Saúde Familiar
USA	United States of America

Chapter 1

Introduction

In this chapter, the problem is overviewed, the study's importance is explained along with goals for the proposed solution.

1.1 Background

Regular expressions are a foundational tool in computer science, widely used in pattern matching, lexical analysis, input validation, and string processing. Their expressiveness and concise syntax make them a powerful language for describing regular languages.

A regular expression R is used (along with an input W) in regex matching engines. The matching engines will verify if W is fully matched by R, meaning that the entire input is a match - or they will verify if a substring of W is matched by R.

1.2 Regular Expression Denial of Service

One such vulnerability is known as *Regular Expression Denial of Service* (ReDoS). ReDoS exploits the pathological worst-case behavior of certain regular expressions, causing exponential time complexity during matching. In typical backtracking matchers—such as those found in JavaScript, Java, and many scripting environments—ambiguous or nested expressions (especially involving repetition, such as `(a+)+`) can lead the engine to explore an exponential number of paths for certain crafted inputs. This behavior allows an attacker to intentionally supply inputs that force excessive computation, effectively rendering a service unavailable or degraded.

The root of the ReDoS problem lies not in regular expressions as a theoretical model but in how they are operationalized in software. While deterministic finite automata (DFAs) evaluate regular expressions in linear time, many real-world engines opt for backtracking NFAs due to their flexibility and ease of implementation. Unfortunately, these NFAs are susceptible to exponential blow-up in ambiguous or unguarded patterns.

1.3 Case Studies

1.3.1 Stack Overflow

On the 20th of July 2016, a user published an malformed post on the online information exchange forum *Stack Overflow.* A couple minutes after the post, at around 14:44 UTC, the entire website became unavailable for around 34 minutes, after which there was an update that fixed the underlying issue.

On the forum, there is an automatic text formatter that runs every time someone posts something. This tool will trim unicode spaces from the start of a line until its end. The regular expression responsible for matching these spaces was the following:

$$\hat{}[\backslash s \backslash u200c] + |[\backslash s \backslash u200c] + \$$$

- ˆ will anchor the following expression to the start of a string
- $[\backslash s\backslash u200c]+$ will match either one whitespace character (space, tab, newline, etc..) or the unicode character U+200C, also known as the *Zero Width Non-Joiner* | is the union operator, meaning that the expression will match either the left or right side expressions (relative to this operator)
- \$ will anchor the match to the end of that string

The automatic text formatter contains a matcher that tried to match the regular expression described above against an input that contained around 20,000 consecutive whitespace characters on one line that started with $--$. The backtracking matcher in place worked as follows:

Given an input string M of length $\mathrm{len}(M)$, let n_k denote the character at position k, where $0 \leq k < \mathrm{len}(M)$. For each possible starting position p such that $0 \leq p < \mathrm{len}(M)$, perform the following steps:

1. Check whether the character n_k (for $k \geq p$) belongs to either the character class `\s` (whitespace) or the Unicode character `\u200c` (zero-width non-joiner).
2. Continue checking characters until a character not in the above classes is encountered or the end of the string is reached.
3. If the end of the string is reached and all characters from position p onward matched, the pattern succeeds.
4. If a non-matching character is encountered before the end of the string, the match fails at this position; increment p and repeat from step 1.

For a 20,000 whitespace-character input, the sum of computations is given as follows:

$$\sum_{k=1}^{20{,}000} k = \frac{20,000 \cdot (20,000) + 1}{2} = 200,010,000$$

This means that this matching algorithm ran in $O(n^2)$ complexity, and this blow up was to be expected.

Chapter 2

Preliminaries

Theory builds upon theory, therefore it is essential to establish a solid foundation by understanding the basic concepts and terminology that compose the core topics of formal languages and automata theory. In this chapter we begin by formally defining what a language is and then move on to describe the class of languages known as regular languages. Along the way, we will also introduce various concepts such as finite automata (DFA, NFA) and regular expressions.

2.1 Alphabets, Strings and Languages

Alphabets

An *alphabet* is a finite, non-empty set of symbols, typically denoted by the Greek letter Σ. That is,

$$\Sigma = \{a_1, a_2, \ldots, a_n\}$$

where each a_i is a symbol in the alphabet.

For example, one can represent the binary alphabet as $\Sigma = \{0, 1\}$, or the English alphabet as $\Sigma = \{a, b, c, \ldots, z\}$.

Strings

A *string* over an alphabet Σ is a finite sequence of symbols from Σ. Strings are typically denoted by w, and the *length* of a string w is denoted by $|w|$.

The set of all strings over the alphabet Σ is denoted by Σ^* and defined as:

$$\Sigma^* = \{w \mid w \text{ is a finite sequence of symbols from } \Sigma\}$$

The unique string of length zero is called the *empty string*, denoted by ε. It is important to note that $\varepsilon \in \Sigma^*$.

For example, if $\Sigma = \{0, 1\}$, then we have that:

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \ldots\}$$

Where the empty string is, as mentioned above, denoted by ε and also belongs to Σ^*.

Languages

A *language* over an alphabet Σ is a set of strings over Σ.

$$L \subseteq \Sigma^*$$

That is, a language is any subset of Σ^*, possibly infinite, finite, or even empty.
Since a language is a set of strings, the following standard set operations can be applied (assuming A and B are languages over the same alphabet Σ):

- *Intersection*: $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$
- *Union*: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
- *Difference*: $A - B = \{x \mid x \in A \text{ and } x \notin B\}$

Furthermore, we can also operate specifically over languages with the following operations (assuming L_1 and L_2 are languages over the same alphabet Σ):

- *Concatenation*: $L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$
- *Kleene Star*: $L^* = \bigcup_{n=0}^{\infty} L^n$, where $L^0 = \{\varepsilon\}$ and $L^n = L \cdot L^{n-1}$ for $n > 0$.
- *Reversal*: $L^R = \{x^R \mid x \in L\}$, where x^R denotes the reversal of string x.
- *Complement*: $\overline{L} = \Sigma^* - L$, i.e., the set of all strings over Σ that are not in L.

These operations form the basis for reasoning about the expressiveness and closure properties of language classes such as regular, context-free, and context-sensitive languages. In particular, regular languages are closed under all the operations listed above, including union, intersection, concatenation, and Kleene star. This robustness makes them especially amenable to algorithmic manipulation, as seen in finite automata and regular expression engines.

2.2 Finite Automata

A *finite automaton* is a theoretical machine used to recognize regular languages. It processes input strings symbol by symbol and determines whether the string belongs to the language defined by the automaton. There are two main types of finite automata:

- **Deterministic Finite Automaton (DFA)**: An automaton where, for each state and input symbol, there is exactly one possible next state.
- **Non-deterministic Finite Automaton (NFA)**: An automaton that allows multiple possible transitions for a given state and input symbol, including transitions without consuming any input (called ε-transitions).

Formally defined, an NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states,
- Σ is the input alphabet,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ is the transition function,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting (final) states.

A string $w \in \Sigma^*$ is accepted by the NFA if there exists a sequence of transitions (possibly including ε-moves) that consumes w and ends in a state q such that $q \in F$.

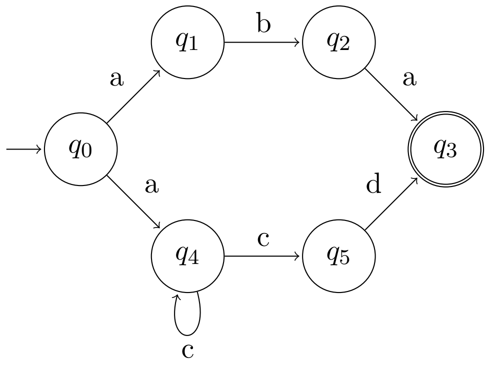

$$\begin{tikzcd} & q_1 \arrow[r, "b"] & q_2 \arrow[dr, "a"] & \\ \arrow[r] & q_0 \arrow[u, "a"] \arrow[d, "a"'] & & q_3 \\ & q_4 \arrow[r, "c"] \arrow[loop below, "c"] & q_5 \arrow[ur, "d"'] & \end{tikzcd}$$

Figure 2.1: Example of an NFA that accepts $L = \{aba, a(c^*)d\}$

An NFA is *deterministic* (also known as DFA) if $|\delta(q, \sigma)| \leq 1$, for any $(q, \sigma) \in Q \times \Sigma$.

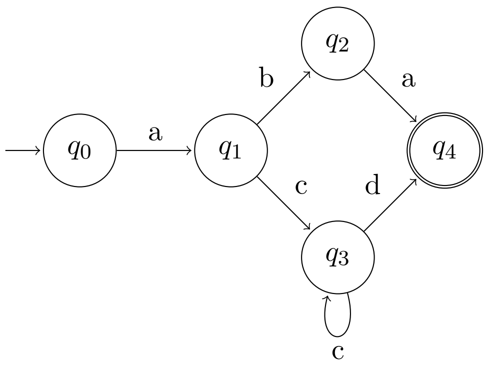


Figure 2.2: A DFA whose language is the same as the NFA from 2.1

2.3 Regular Expressions

Let Σ be a finite alphabet. Let $L \subseteq \Sigma$. The set of *regular expressions* over Σ, denoted by RegExp(Σ), is defined inductively as follows:

- $\emptyset$ is a regular expression denoting the empty language: $L(\emptyset) = \emptyset$.
- ε is a regular expression denoting the language containing only the empty string: $L(\varepsilon) = \{\varepsilon\}$.
- For each symbol $a \in \Sigma$, a is a regular expression denoting the singleton language: $L(a) = \{a\}$.
- If r_1 and r_2 are regular expressions, then so are:
  - $(r_1 \mid r_2)$, denoting the union: $L(r_1 \mid r_2) = L(r_1) \cup L(r_2)$.
  - $(r_1 \cdot r_2)$, denoting concatenation: $L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2)$.
  - $(r_1)^*$, denoting Kleene star: $L(r_1^*) = (L(r_1))^*$.

We write RegExp(Σ) to denote the set of all such syntactic expressions, and for each $r \in$ RegExp(Σ), the function $L(r)$ yields the language defined by r.

Parentheses are used to disambiguate expressions and enforce precedence; by convention, Kleene star binds most tightly (a^*), followed by concatenation (e.g. $a \cdot b$, whose operator "$\cdot$" is omitted for convenience), and finally union ($+$).

One can see if the regular expression contains an empty word by utilizing the *empty word property.* The property is defined as the function $\varepsilon : \mathit{RegExp} \rightarrow \{\varepsilon, \emptyset\}$.

Given $\alpha, \beta \in \text{RegExp}(\Sigma)$ and $a \in \Sigma$,

$$\varepsilon(\emptyset) = \emptyset$$
$$\varepsilon(\varepsilon) = \varepsilon$$
$$\varepsilon(a) = \emptyset$$

2.3.1 Extended Regular Expressions

In addition to the basic operations, some extended operators are often used for convenience. These include:

- **Kleene plus**: Given a regular expression r, the expression r^+ denotes one or more repetitions of r:
$$L(r^+) = L(r) \cdot L(r)^*.$$
- **Fixed repetition (power)**: For a regular expression r and integer $n \geq 0$, the expression r^n denotes n consecutive concatenations of r:
$$L(r^0) = \{\varepsilon\}, \quad L(r^n) = L(r) \cdot L(r^{n-1}) \text{ for } n > 0.$$
- **Bounded repetition**: For a regular expression r and integers m, n with $0 \leq m < n$, the bounded repetition $r^{[m,n[}$ denotes the language containing all strings formed by concatenating between m and n copies of strings from $L(r)$:
$$L(r^{[m,n[}) = \bigcup_{k=m}^{n} L(r^k).$$

These extended forms do not increase the expressive power of regular expressions but are useful for readability and practical applications. They can always be rewritten using the fundamental operators: union and concatenation.

2.3.2 Derivatives

The *derivative of a regular expression* was first introduced in 1962 by Janusz Brzozowski. It is a powerful concept used to define the behavior of regular expressions in a more operational manner. They can be used as a means of verifying equivalence of regular expressions, for example. The derivative of a regular expression r with respect to a symbol a is another regular expression $D_a(r)$ that describes the set of strings that can be obtained by taking the derivative of r with respect to a.

The derivative of a regular expression r with respect to $\sigma \in \Sigma$ is itself a regular expression $d_\sigma(r)$ such that $\mathcal{L}(d_\sigma(r)) = \{w \mid \sigma w \in \mathcal{L}(r)\}$ is defined as:

$$\begin{aligned}
d_\sigma(\emptyset) &= \emptyset \\
d_\sigma(\varepsilon) &= \emptyset \\
d_\varepsilon(r) &= r \\
d_\sigma(r') &= \begin{cases} \varepsilon & \text{if } \sigma' = \sigma \\ \emptyset & \text{otherwise,} \end{cases} \\
d_\sigma(r + r') &= d_\sigma(r) + d_\sigma(r') \\
d_\sigma(rr') &= \begin{cases} d_\sigma(r)r' & \text{if } \varepsilon(r) = \emptyset \\ d_\sigma(r)r' + d_\sigma(r') & \text{otherwise,} \end{cases} \\
d_\sigma(r^*) &= d_\sigma(r)r^* \\
d_\sigma(r^+) &= d_\sigma(r)r^*
\end{aligned}$$

Brzozowski defined a DFA equivalent to a regular expression with the help of derivatives. With this, it is important to note that, for example, a regular expression $r = a^*$ (matches zero or more 'a' symbols) can be used to construct an equivalent DFA using Brzozowski's construction, even though $d_a(r) = a \cdot d_a(r)$ which can lead to an infinite construction. This example is enough to show that the set of all derivaives of a regular expression may not be finite.

2.3.3 Partial Derivatives

The notion of *partial derivative* was introduced by Antimirov. Opposite to Brzozowski's derivative, partial derivatives can lead to the construction of an NFA.

Let $\alpha \in \mathit{RegExp}(\Sigma)$ be a *regular expression* over Σ. The set of partial derivatives of α with respect to a symbol $b \in \Sigma$, represented as $\partial_b(\alpha)$, is defined as follows:

$$\begin{aligned}
\partial_b(\emptyset) &= \emptyset & \partial_b(\alpha + \beta) &= \partial_b(\alpha) \cup \partial_b(\beta),\ \beta \neq \alpha \\
\partial_b(\varepsilon) &= \emptyset & \partial_b(\alpha\beta) &= \partial_b(\alpha)\beta \cup \partial_b(\beta) \\
\partial_b(b) &= \varepsilon & \partial_b(\alpha\beta) &= \delta_b(\alpha)\beta \\
\partial_b(c) &= \emptyset,\ b \neq c \text{ and } b \in \Sigma & \partial_b(\alpha^*) &= \partial_b(\alpha)\alpha^*
\end{aligned}$$

2.4 From Regular Expressions to Automata

While regular expressions provide a declarative way to specify patterns in strings, finite automata offer an operational model for recognizing such patterns.

2.4.1 Thompson's Algorithm

Thompson's construction is a classic algorithm for converting a regular expression into a nondeterministic finite automaton (NFA) with ε-transitions. It was introduced in 1968 by Ken Thompson and is the foundation for many regex engines, including the `lex` lexical analyzer generator.

The construction proceeds recursively based on the structure of the regular expression. Each base case (such as a single symbol or the ε) and each operator ('+', concatenation, '*') corresponds to a small NFA fragment. These fragments are then joined using ε-transitions.

Although Thompson's NFA may contain many ε-transitions, it is guaranteed to be of size linear in the length of the regular expression. The resulting NFA can be converted into a deterministic finite automaton (DFA) using the standard powerset construction, typically after removing ε-transitions.

2.4.2 Brzozowski's Derivatives

Through the definition of *derivatives* mentioned in 2.3.2, one can compute the derivatives for all symbols, memoize the results and build a DFA that represents the same language as the intended regular expression.

This method can, however, lead to an exponential number of distinct derivatives in the worst case. Therefore, simplification rules and expression equivalences are critical to making the approach practical.

2.4.3 Antimirov's Partial Dertivatives

Proposed by Valery Antimirov in 1996, the partial derivatives construction generalizes Brzozowski's derivatives to build an NFA rather than a DFA. Instead of producing a single derivative for each symbol, Antimirov's method produces a *set* of partial derivatives, reflecting the inherent nondeterminism of the regular expression.

This construction avoids ε-transitions and yields a compact ε-free NFA. Each partial derivative corresponds to a transition in the automaton, and the process naturally handles alternation and repetition.

Antimirov's approach is especially efficient for regex evaluation and analysis tasks, and forms the basis for several modern regex matchers and formal verification tools.

2.4.4 Position Automata

The *position automaton*, also known as the *Glushkov automaton*, is a type of ε-free nondeterministic finite automaton (NFA) constructed directly from a regular expression. Unlike the standard Thompson construction, which introduces ε-transitions that must later be eliminated, the Glushkov construction yields an automaton in which each state corresponds uniquely to a symbol occurrence—or *position*—in the expression. [1]

Given a regular expression E, the Glushkov automaton M_E is defined based on three key position-based functions:

- **first**(E): the set of positions that can appear first in some word of the language $\mathcal{L}(E)$.
- **last**(E): the set of positions that can appear last in some word of $\mathcal{L}(E)$.
- **follow**(E, x): for each position x, the set of positions that can immediately follow x in some word of $\mathcal{L}(E)$.

To distinguish different occurrences of the same symbol, the construction introduces marked symbols. For example, the expression $(a+b)^*a(b+a)^*$ is rewritten as $(a_1+b_2)^*a_3(b_4+a_5)^*$. Each marked symbol corresponds to a unique position and becomes a distinct state in the automaton.

The Glushkov automaton $M_E = (Q, \Sigma, \delta, q_0, F)$ is constructed as follows:

- Q is the set of positions in E (i.e., the marked symbols), plus an initial state q_0.
- For each symbol $a \in \Sigma$:
  - $\delta(q_0, a) = \{x \in \text{first}(E) \mid \text{symbol}(x) = a\}$
  - $\delta(x, a) = \{y \in \text{follow}(E, x) \mid \text{symbol}(y) = a\}$
- The set of final states F is last(E); if $\varepsilon \in \mathcal{L}(E)$, then q_0 is also final.

This automaton captures the structural flow of E by tracing symbol sequences as state transitions. It is ε-free and has one state per symbol occurrence, which results in at most a quadratic number of transitions with respect to the size of E.

An important property of the Glushkov automaton is its relationship to unambiguity. A regular expression is *weakly unambiguous* if and only if its Glushkov automaton is unambiguous, i.e., there exists at most one accepting path for each accepted word. This makes the Glushkov construction a practical and efficient tool in applications requiring unambiguous parsing, such as syntax analysis in document type definitions (e.g., SGML and XML DTDs).

2.5 FAdo

The FAdo [2] project is an open-source implementation of several sets of tools for formal languages manipulation. In order to allow quick prototyping and testing of algorithms, these tools were developed in Python. Regular languages can be represented by regular expressions which are defined in `reex.py` - or by finite automata which are defined in `fa.py`.

2.5.1 Regular Expressions

In `reex.py`, FAdo defines the `RegExp` base class for all regular expressions, declaring therein a class variable `sigma`, formally known as the set of symbols (Σ). To represent any and all base constructions of a regular expressions, FAdo defines base classes for each of them:

- **CEmptySet**: The empty symbol set. ($\emptyset$)
- **CEpsilon**: The empty string. (ς)
- **CAtom**: A simple symbol. (e.g. $'a'$)
- **CDisj**: The + operation between symbols. (e.g. `CDisj(CAtom(a),CAtom(b))` represents the regular expression $R = a + b$ where $a, b \in \Sigma_R$)
- **CConcat**: The $\cdot$ operation
- **CStar**: The Kleene closure over a set of symbols (e.g. `CStar(CDisj(CAtom(a),CAtom(b)))` representes the regular expression $R = (a + b)^*$ where $a, b \in \Sigma_R$)

In order to parse expressions into FAdo's classes and types, *lark* was used. *lark* is a parsing toolkit for Python. It can parse all context-free languages.

2.5.2 Finite Automata

2.5.3 Extended Regular Expressions

Chapter 3

State of the Art

The current state-of-the-art for mitigating the **ReDoS!** What!

Chapter 4

Counting

In this chapter, we will explore the concept of counting in the context of formal languages and automata theory as well as explain an attempt that was made towards match counting using a partial derivative automaton construction and why it didn't work.

4.1 Counting in Formal Languages

In formal language theory, counting refers to the ability of a language or an automaton to enforce numeric constraints over the number of symbols or patterns within strings. Specifically, it deals with the ability to recognize whether certain elements occur a specified number of times—or in a specific numerical relationship to others.

4.2

Chapter 5

Matching

Regex *matching* is the process of checking whether a piece of text fits a specific pattern described using a regular expression (regex). A regex is a compact, rule-based way to describe sets of strings—like email addresses, phone numbers, or specific word formats.

In this chapter, we will discuss the different approaches to matching regular expressions, the implications of using them, and the performance considerations that arise from these choices. Furthermore, we will also present a novel approach based on position automata, which aims to mitigate the performance issues associated with traditional regex engines while preserving some of the extended expressiveness of regex patterns.

5.1 Modified Position Automata

A *position automaton* is a type of nondeterministic finite automaton (NFA) that facilitates overlapped matching.

Algorithm 1 NFAPOSCOUNT(R): Construct Special Position Automaton

```
Require: Regular expression $R$
Ensure: NFA $A$
 1: $A \leftarrow$ new empty NFA
 2: $i \leftarrow A$.addInitialState()
 3: $A$.addTransitionStar($i, i$)                    ▷ Accept any symbol from $\Sigma$
 4: $f_R \leftarrow R$.marked()
 5: stack $\leftarrow$ empty stack
 6: addedStates $\leftarrow$ empty map
 7: for all $p \in First(f_R)$ do
 8:     $q \leftarrow A$.addState($p$)
 9:     addedStates[$p$] $\leftarrow q$
10:     stack.push(($p, q$))
11:     $A$.addTransition($i, p, q$)
12: end for
13: while stack is not empty do
14:     $(s, s_{\text{idx}}) \leftarrow$ stack.pop()
15:     for all $t \in Follow(f_R, s)$ do
16:         if $t \in$ addedStates then
17:             $q \leftarrow$ addedStates[$t$]
18:         else
19:             $q \leftarrow A$.addState($t$)
20:             addedStates[$t$] $\leftarrow q$
21:             stack.push(($t, q$))
22:         end if
23:         $A$.addTransition($s_{\text{idx}}, t, q$)
24:     end for
25: end while
26: $e \leftarrow A$.addState()
27: $A$.addTransitionStar($e, e$)
28: for all $p \in f_R$.Last() do
29:     if $p \in$ addedStates then
30:         $A$.addFinal(addedStates[$p$])
31:         $A$.addTransitionStar(addedStates[$p$], $e$)
32:     end if
33: end for
```

5.2 Automata-Based Matching

Given a regular expression R, one can construct an NFA A such that $L(A) = L(R)$. Matching then reduces to verifying whether the automaton A accepts the input string s. In DFA-based engines, each character of the input leads to a deterministic transition from one state to another, resulting in a guaranteed linear-time match. In contrast, NFA-based engines may involve branching paths due to nondeterminism and can require simulating multiple transitions concurrently.

For example, consider the following regex pattern:

```
^(a+)+$
```

5.3 Matching with the Modified Position Automaton

One can find all matches over an input string by constructing the modified position automaton from a regular expression and then simulating the automaton's transitions over the input string. The algorithm presented in this section is designed to track the start and end positions of all matches, including overlapping ones, without relying on backtracking.

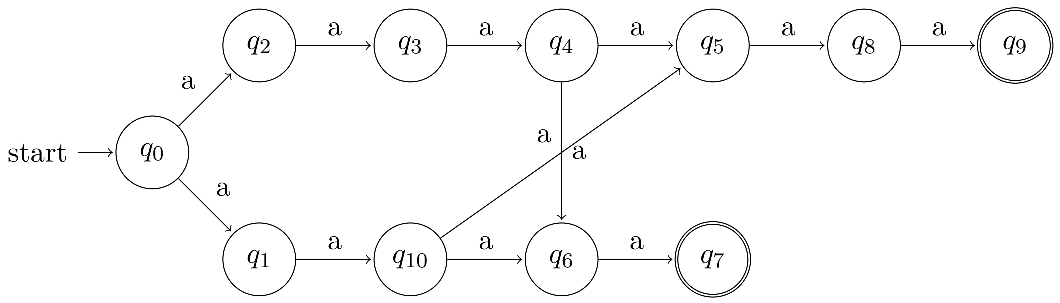
Algorithm 2 TABLEMATCHER(A, s): Modified Position Automaton Multi-matcher

```
Require: $A = (\Sigma, Q, \delta, I, F)$: NFA
Require: $s$: input string
Ensure: $M$: mapping from final states to lists of match positions
 1: $symbols \leftarrow$ list with $\varepsilon$ prepended to $s$
 2: $currentRow \leftarrow$ empty map from states to list of position pairs
 3: $finalMatches \leftarrow$ empty map from states to list of matches
 4: $position \leftarrow 0$
 5: for all $sym$ in $symbols$ do
 6:     if $sym = \varepsilon$ then
 7:         for all $q_0 \in I$ do
 8:             $currentRow[q_0] \leftarrow [(0, 0)]$
 9:         end for
10:     else
11:         $nextRow \leftarrow$ empty map
12:         if $sym \in \Sigma$ then
13:             for all $q \in$ keys($currentRow$) do
14:                 if $|\delta(q, sym)| > 0$ then
15:                     for all $q' \in \delta(q, sym)$ do
16:                         for all $(start, \_) \in currentRow[q]$ do
17:                             if $q' = q$ and $q' \in I$ then
18:                                 append $(position, position)$ to $nextRow[q']$
19:                             else
20:                                 append $(start, position)$ to $nextRow[q']$
21:                                 if $q' \in F$ then
22:                                     append $(start, position)$ to $finalMatches[q']$
23:                                 end if
24:                             end if
25:                         end for
26:                     end for
27:                 end if
28:             end for
29:         else                          ▷ Symbol not in $\Sigma$; treat as fresh start
30:             for all $q_0 \in I$ do
31:                 $nextRow[q_0] \leftarrow [(position, position)]$
32:             end for
33:         end if
34:         $currentRow \leftarrow nextRow$
35:         $position \leftarrow position + 1$
36:     end if
37: end for
38: return $finalMatches$
```

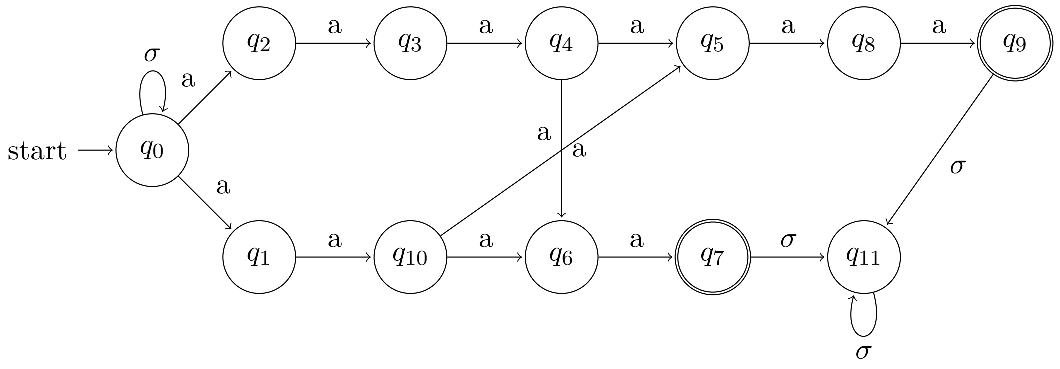
As an example, consider the following regular expression R and input string w:

$$R = (aa + aaa)(aaa + aa)$$
$$w = aaaaabaaaaa$$

The Glushkov automaton construction for R is as follows:



Meanwhile, the modified position automaton for this regular expression can be constructed using the algorithm presented in 1, resulting in the following:



One can then apply the matching algorithm (2) to an input string, resulting in the following match table:

Bibliography

[1] S Broda, M Holzer, E Maia, N Moreira, and R Reis. A mesh of automata. *Information and Computation*, 265:94–111, 2019. doi:10.1016/j.ic.2019.01.003.

[2] Rogério Reis and Nelma Moreira. Fado: tools for finite automata and regular expressions manipulation. 11 2002.