Abstract

Regular expressions (*regex*) formalize a class of patterns definable over strings, corresponding to the family of regular languages in automata theory. They provide a declarative mechanism for specifying sets of strings, making them central both to theoretical models of computation and to practical applications such as 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#* [6] and *Hyperscan* [8].

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

DFA Deterministic Finite Automaton

FCUP Faculdade de Ciências da Universidade do Porto

NFA Non-deterministic Finite Automaton

ReDoS Regular Expression Denial of Service

RegEx Regular Expression

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 running 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 only in regular expressions as a theoretical model but also 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

In this section, two case studies are presented. These serve as a form of introduction to getting to know and understand the ReDoS problem.

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 any group of leading whitespace or invisible characters at both the beginning and end of a post. The regular expression that does so is the following:

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

- $\hat{}$ will anchor the following expression to the start of a string
- $[\backslash s\backslash u200c]+$ will match one or more of either whitespace characters (space, tab, newline, etc..) or the unicode characters U+200C (zero-width non-joiner)
- \$ will anchor the match to the end of that string

The aforementioned tool was an automatic text formatter which containined 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 $\text{len}(M)$, let n_k denote the character at position k, where $0 \leq k < \text{len}(M)$. For each possible starting position p such that $0 \leq p < \text{len}(M)$, perform the following steps:

1. Check whether the character n_k (for $k \geq p$) is either a whitespace or a zero-width non-joiner (U+200C).
2. Continue checking characters until it reaches a character that is neither a whitespace nor a zero-width non-joiner.
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 (both whitespace and zero-width non-joiner characters) 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 the matching algorithm ran in $O(n^2)$ complexity, and this blow up was to be expected.

The engineers quickly fixed this issue by switching to a substring replacing method.

1.3.2 minimatch

minimatch is a minimal matching utility, used internally by the **Node Package Manager**, better known as **npm**. [9] This utility works by convering glob expressions into JavaScript's *RegExp* objects, supporting the following glob features:

- Brace Expansion
- Extended glob matching
- "Globstar" (**) matching
- POSIX character classes

The utility has millions of downloads, as it is an essential component of **npm**.

On the 18th of October 2022, a new CVE was introduced: **CVE-2022-3517**. The report showed that all of minimatch's versions below 3.0.5 were vulnerable to a ReDoS attack similar to the one described in 1.3.1. The culprit for this was a function called *braceExpand*, which is responsible for expanding brace patterns in glob strings. This is commonly known as brace expansion and is often seen in Unix shells (such as bash). The function contained a regular expression that would match against given patterns and decide if a brace expansion was in order. The expression used was "/\{.*\}/)", which matches any string containing a single pair of curly braces with any characters inside. But this expression poses an issue:

For example, the following text:

"{{{{{{{{{{{{{...X"

with the { repeated over 30,000 times and no closing }, can cause a significant CPU spike or even hang the process due to catastrophic backtracking.

To fix this issue, the developer decided to switch to a safer regular expression: \{(?:(?!\{).)*\}/

1.4 Reluctance Toward Changing Legacy Matchers

Despite well-documented vulnerabilities such as ReDoS, there remains significant reluctance in the software engineering community to replace or refactor legacy regex engines—particularly those built into performance-critical or widely adopted tools such as `grep`, `sed`, and many scripting languages.

These tools often rely on matching engines that prioritize speed and simplicity of implementation over safety. For example, `grep` and similar UNIX utilities implement regex matchers using finite automata, but their behavior with extended features (like bounded repetitions) can still lead to performance degradation in edge cases. While these engines are generally immune to the exponential blow-up typical of backtracking matchers, they may suffer from linear but high-cost processing when automata grow excessively large due to poorly constructed patterns.

The situation is more severe in environments that rely on backtracking matchers, such as JavaScript, Java, and many shell-based text processors. In these ecosystems, regular expressions are both expressive and dangerously permissive, allowing patterns that trigger catastrophic backtracking without warning.

Refactoring or replacing these engines is often resisted for several reasons:

- **Backwards compatibility**: Legacy codebases and systems expect specific regex semantics, and changing the underlying engine could break existing behavior.
- **Perceived performance cost**: DFA-based matchers may require significant memory or preprocessing, which is viewed as a performance risk in lightweight tools.
- **Lack of awareness**: Many developers are unaware that regex matchers can introduce denial-of-service vulnerabilities, especially when ReDoS exploits are subtle or input-driven.
- **Cultural inertia**: Tools like `grep` are deeply embedded in developer workflows and scripts, making any modification to their behavior or performance profile controversial.

Even modern matchers that are designed to avoid ReDoS—such as Google's RE2 or Rust's regex crate—are often underutilized due to these legacy constraints.

These factors highlight the need for not only technical solutions, such as safer regex engines and static analysis tools, but also a cultural shift in how regular expressions are authored, reviewed, and validated in production systems.

The next chapter will give some more insight into the basics of regular expressions and automata.

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^*.

The concatenation of two strings w and v over Σ is denoted by wv (or, more explicitely, $w \cdot v$) and results in a separate string composed by the singular string v right after w. For instance, let $w = ab$ and $v = cd$, the concatenation wv is $abcd$. We can define the concatenation of two words $w, v \in \Sigma^*$ as follows:

$$w \cdot \varepsilon = \varepsilon \cdot w = w,$$
$$(wa) \cdot v = w \cdot (av), a \in \Sigma.$$

Given a string $w = a_1, a_2, ..., a_n$, the reversal of w, denoted by w^R, is the word formed by reversing the order of its characters such that $w^R = a_n, a_{n-1}, ..., a_1$. We can define the reversal of a string recursively like so:

$$w^R = v^R a, \quad \text{where } w = av, a \in \Sigma \text{ and } v \in \Sigma^*,$$
$$\varepsilon^R = \varepsilon.$$

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*: $A \setminus B = \{x \mid x \in A \text{ and } x \notin B\}$

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 a 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 this instance, the class of regular languages over an alphabet Σ is closed under union ($\bigcup$), concatenation ($\cdot$) 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 model of computation 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,
- $Q_0 \subseteq Q$ is the set of initial states,
- $F \subseteq Q$ 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 $q \in F$.

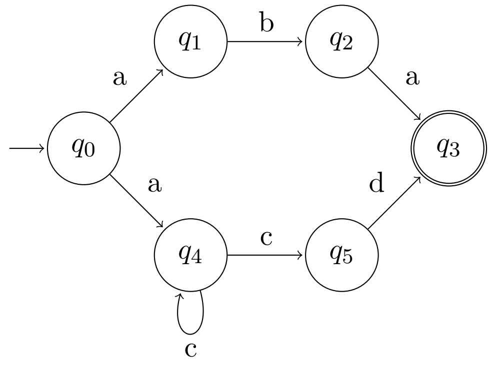

Figure 2.1: Example of an NFA that accepts $L = \{aba, a(c^*)d\}$

An NFA is *deterministic* (also known as DFA) if $|\delta(q, \sigma)| \leq 1$, for any $(q, \sigma) \in Q \times \Sigma$ and $|Q_0| = 1$.

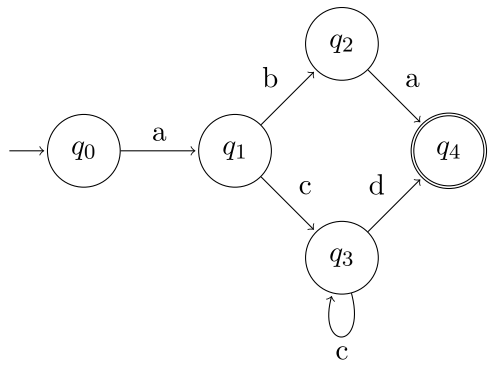

Figure 2.2: A DFA whose language is the same as the NFA from figure 2.1

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:
  - $(r_1 \mid r_2)$ or $(r_1 + r_2)$, 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: $L(r_1^*) = (L(r_1))^*$.

For each $r \in \text{RegExp}(\Sigma)$, 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 ($+$).

We denote by $\alpha = \beta$ if two regular expressions α and β, both over Σ, represent the same language ($L(\alpha) = L(\beta)$).

If the language of a regular expression α contains the empty word, then that regular expression possesses the *empty word property*.

Definition 2.3.1 (Empty Word Property)**.** The empty word property is characterized by the function $\varepsilon : \text{RegExp} \to \{\varepsilon, \emptyset\}$ and is defined recursively as follows, given α and β regular expressions defined over Σ:

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

$$\varepsilon(\varepsilon) = \varepsilon$$

$$\varepsilon(a) = \emptyset,\ a \in \Sigma$$

$$\varepsilon(\alpha + \beta) = \begin{cases} \emptyset & \text{if } \varepsilon(\alpha) = \varepsilon(\beta) = \emptyset \\ \varepsilon & \text{if } \varepsilon(\alpha) = \varepsilon \text{ or } \varepsilon(\beta) = \varepsilon \end{cases}$$

$$\varepsilon(\alpha\beta) = \begin{cases} \emptyset & \text{if } \varepsilon(\alpha) = \emptyset \text{ or } \varepsilon(\beta) = \emptyset \\ \varepsilon & \text{if } \varepsilon(\alpha) = \varepsilon(\beta) = \varepsilon \end{cases}$$

$$\varepsilon(a^*) = \varepsilon$$

2.3.1 Extended Regular Expressions

In addition to the basic operations, some other 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 $0 \leq m \leq n-1$, the bounded repetition $r^{[m,n]}$ denotes the language containing all strings formed by concatenating between m and $n-1$ copies of strings from $L(r)$:
$$L(r^{[m,n]}) = \bigcup_{k=m}^{n-1} L(r^k).$$

These 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.

Definition 2.3.2 (Derivative of a Regular Expression [3])**.** The derivative of a regular expression r with respect to $\sigma \in \Sigma$ is itself a regular expression $d_\sigma(r)$ such that $\mathcal{L}(d_\sigma(r)) = \{w \mid \sigma w \in \mathcal{L}(r)\}$ and is defined as:

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

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 $d_a(r) = a \cdot d_a(r)$ may look like it can lead to an infinite construction.

Definition 2.3.3 ([3])**.** Two regular expressions are similar if one can be transformed to the other using only the identities:

$$
\begin{aligned}
&R + R = R, \\
&P + Q = Q + P, \\
&(P + Q) + R = P + (Q + R) \\
&\varepsilon R = R\varepsilon = R \\
&\emptyset R = R\emptyset = \emptyset
\end{aligned}
$$

Two regular expressions are dissimilar if and only if they are not similar.

With this, Brzozowski proved in [3, Theorem 5.2] that every regular expression has only a finite number of dissimilar derivatives. The big disadvantage is that, as he stated, the resulting automaton is far from minimal.

2.3.3 Partial Derivatives

The notion of *partial derivative* was introduced by Antimirov [1]. Opposite to Brzozowski's derivatives, *partial derivatives* will lead to the construction of an NFA.

Let $\alpha \in RegExp(\Sigma)$ 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:

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

The set of partial derivatives by a word $w \in \Sigma^*$

For instance, given the regular expression $r = (ad + d^*)db$, where $\Sigma = \{a, b, d\}$, the set of partial derivatives of r for the symbol d is given by

$$
\begin{aligned}
\partial_d(r) &= \partial_d((ad + d^*)db) \\
&= \partial_d(addb + d^*db) \\
&= \partial_d(addb) \cup \partial_d(d^*db) \\
&= \partial_d(a)ddb \cup \partial_d(d^*)db \cup \partial_d(db) \\
&= \emptyset ddb \cup \partial_d(d)d^*db \cup \partial_d(d)b \cup \partial_d(b) \\
&= \emptyset \cup d^*db \cup b \cup \emptyset
\end{aligned}
$$

resulting in $\partial_d(r) = \{d^*db, b\}$.

2.3.3.1 Linear Form

It is possible to calculate the partial derivatives for every symbol $a \in \Sigma$ of a regular expression r over Σ in a single pass. To do this, Antimirov [1] defined the *linear form*.

A linear form lf of a regular expression α is a finite set of symbol–continuation pairs. These pairs are called *monomials* and they can encode all the possible one-symbol prefixes of that regular expression.

Formally, given an alphabet Σ and the set of all regular expressions over Σ denoted *RegExp*, a monomial is an element of $\Sigma \times \mathit{RegExp}$. We denote the set of all monomials for a given alphabet Σ by M_Σ.

A *linear form* is then a finite set of monomials:

$$lf(\alpha) \subseteq 2^{M_\Sigma}.$$

For instance, a linear form $l = \{(a_1, \alpha_1), \ldots, (a_n, \alpha_n)\}$ corresponds to the regular expression

$$\kappa_l \ = \ a_1 \cdot \alpha_1 \ + \ \cdots \ + \ a_n \cdot \alpha_n.$$

The function $lf : \mathit{RegExp} \to 2^{M_\Sigma}$ is defined recursively as follows, where $\alpha, \beta \in \mathit{RegExp}$:

$$\begin{aligned}
lf(\emptyset) &= \emptyset \\
lf(\varepsilon) &= \emptyset \\
lf(a) &= \{(a, \varepsilon)\}, \quad a \in \Sigma \\
lf(\alpha + \beta) &= lf(\alpha) \cup lf(\beta) \\
lf(\alpha \cdot \beta) &= (lf(\alpha) \cdot \beta) \ \cup \ \begin{cases} lf(\beta) & \text{if } \varepsilon \in L(\alpha) \\ \emptyset & \text{otherwise} \end{cases} \\
lf(\alpha^*) &= lf(\alpha) \cdot \alpha^*
\end{aligned}$$

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

In 1968, Thompson provided a method capable of transforming a regular expression into an equivalent nondeterministic finite automaton with ε-transitions (an ε-NFA) [5]. The main idea was to associate each regular-expression operator with a small automaton fragment, and then combine these fragments recursively until the entire expression is represented.

Each basic symbol $a \in \Sigma$ is represented by two states with a transition labeled a between them. The empty string ε is represented by two states connected by a single ε-transition.

More complex expressions are built by combining smaller fragments:

- Concatenation ($\alpha\beta$) is obtained by connecting the accept state of α to the start state of β with an ε-transition.

- Union ($\alpha \mid \beta$) is built by adding a new start state with ε-transitions to the start states of α and β, and a new accept state with ε-transitions from the accept states of α and β.
- Kleene star (α^*) is represented by adding a new start and accept state. The new start connects by ε both to the new accept (for the empty repetition) and to the start of α (for one or more repetitions). The accept state of α connects by ε back to its own start and to the new accept.

Thompson described this as a compiler-like process: the expression is first parsed into a convenient form (such as reverse Polish notation), and then a stack-based procedure builds the automaton fragment by fragment.

In the end, a single fragment represents the entire regular expression. The resulting automaton can be simulated efficiently by maintaining two lists of active states: the current list and the next list. For each input symbol, all ε-reachable states are added to the current list, transitions on the current symbol are followed, and the next list becomes the current list for the following step. A string is accepted if the accept state becomes active at the end of the input. An example of an automaton constructed using Thompson's method is shown in 2.3.

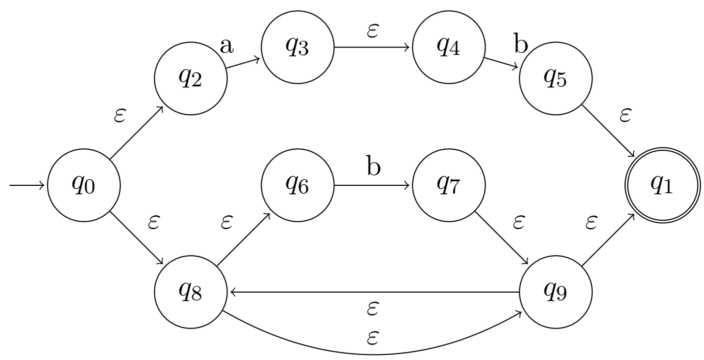

Figure 2.3: Diagram of a ε-NFA built using Thompson's construction for the regular expression $(ab + b^*)$.

2.4.2 Brzozowski's Derivatives

In 1964, Brzozowski [3] proposed a method of constructing deterministic finite automata using derivatives, where consequent derivations of a regular expression would result in the states and their transitions with respect to the symbol derived.

Definition 2.4.1 (Brzozowski's Automaton [3])**.** Let $\alpha \in \mathbf{RegExp}$ be a regular expression defined over Σ. Brzozowski's automaton for the regular expression α is defined as the deterministic finite automaton $\mathcal{D} = (Q, \Sigma, q_0, \delta, F)$. Q is the set of all dissimilar derivatives of α, q_0 is the initial state ($q_0 \in Q$), δ is the transition function defined by $\delta : Q \times \Sigma \to Q$ such that $\delta(q, a) = d_a(q)$ for all $a \in \Sigma$ and $q \in Q$, and the set of final states is $F = \{q \in Q \mid \varepsilon(q) = \varepsilon\}$.

For instance, let $\alpha = (ad + d^*)db$ over $\Sigma = \{a, b, d\}$. The transition function δ is defined for a Brzozowski's automaton $\mathcal{D}$ of α as:

$\alpha = (ad + d^*)db$:

$$\begin{aligned}
\delta((ad + d^*)db, a) &= d_a((ad + d^*)db) \\
&= d_a(addb + d^*db) \\
&= d_a(addb) + d_a(d^*db) \\
&= d_a(a)ddb + \emptyset + \emptyset \\
&= \varepsilon ddb
\end{aligned}$$

$$\begin{aligned}
\delta((ad + d^*)db, b) &= d_b((ad + d^*)db) \\
&= d_b(addb + d^*db) \\
&= d_b(addb) + d_b(d^*db) \\
&= \emptyset + \emptyset = \emptyset
\end{aligned}$$

$$\begin{aligned}
\delta((ad + d^*)db, d) &= d_d((ad + d^*)db) \\
&= d_d(addb + d^*db) \\
&= d_d(addb) + d_d(d^*db) \\
&= \emptyset + d_d(d^*)db + d_d(db) \\
&= d_d(d)d^*db + d_d(d)b + d_d(b) \\
&= \varepsilon d^*db + \varepsilon b + \emptyset \\
&= d^*db + b
\end{aligned}$$

$\alpha = ddb$:

$$\begin{aligned}
\delta(ddb, a) &= d_a(ddb) \\
&= d_a(d)db \\
&= \emptyset
\end{aligned}$$

$$\begin{aligned}
\delta(ddb, b) &= d_b(ddb) \\
&= d_b(d)db \\
&= \emptyset
\end{aligned}$$

$$\begin{aligned}
\delta(ddb, d) &= d_d(ddb) \\
&= d_d(d)db \\
&= \varepsilon db \\
&= db
\end{aligned}$$

$\alpha = d^*db + b$:

$$\begin{aligned}\delta(d^*db + b, a) &= d_a(d^*db + b)\\ &= d_a(d^*db) + d_a(b)\\ &= \emptyset + \emptyset = \emptyset\end{aligned}$$

$$\begin{aligned}\delta(d^*db + b, b) &= d_b(d^*db + b)\\ &= d_b(d^*db) + d_b(b)\\ &= \emptyset + \varepsilon = \varepsilon\end{aligned}$$

$$\begin{aligned}\delta(d^*db + b, d) &= d_d(d^*db + b)\\ &= d_d(d^*db) + d_d(b)\\ &= d_d(d^*)db + d_d(db) + \emptyset\\ &= d_d(d)d^*db + d_d(d)b + d_d(b)\\ &= \varepsilon d^*db + \varepsilon b + \emptyset = d^*db + b\end{aligned}$$

$\alpha = db$:

$$\begin{aligned}\delta(db, a) &= d_a(db)\\ &= d_a(d)b\\ &= \emptyset\end{aligned}$$

$$\begin{aligned}\delta(db, b) &= d_b(db)\\ &= d_b(d)b\\ &= \emptyset\end{aligned}$$

$$\begin{aligned}\delta(db, d) &= d_d(db)\\ &= d_d(d)b\\ &= \varepsilon b = b\end{aligned}$$

$\alpha = b$

$$\begin{aligned}\delta(b, a) &= d_a(b)\\ &= \emptyset\end{aligned}$$

$$\begin{aligned}\delta(b, b) &= d_b(b)\\ &= \varepsilon\end{aligned}$$

$$\delta(b, d) = d_d(b)$$
$$= \emptyset$$

Finally, the set of states for $\mathcal{D}$ is $Q = \{(ad + d^*)db, ddb, d^*db + b, db, b, \varepsilon, \emptyset\}$ and the set of final states is $F = \{\varepsilon\}$. The diagram for this automaton is shown in 2.4.

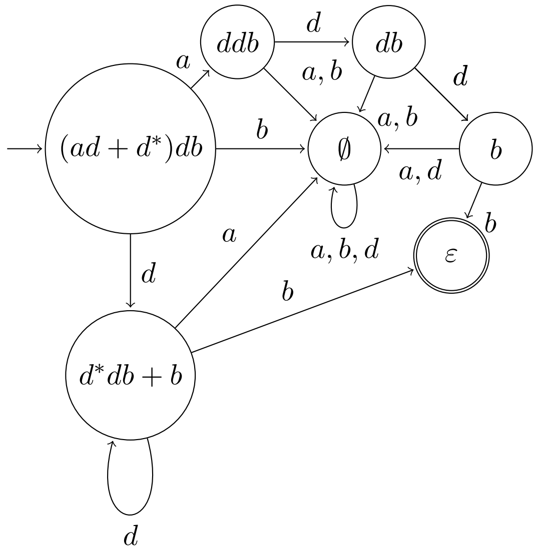

Figure 2.4: Diagram of a Brzozowki DFA for the regular expression $(ad + d^*)db$.

2.4.3 Antimirov's Partial Derivatives

Proposed by Valery Antimirov in 1996 [1], 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.

Definition 2.4.2 (Partial Derivatives Automaton)**.** Let $\alpha \in \mathit{RegExp}$ over Σ. The partial derivatives automaton for α is defined as the non-deterministic finite automaton $\mathcal{N} = (Q_{\mathcal{PD}}, \Sigma, q_0, \delta, F)$. q_0 is the initial state and belongs to the set of states $Q_{\mathcal{PD}}$. It is important to note that $q_0 = \alpha$. The transition function is defined as $\delta(q, a) = \partial_a(q)$ for all symbols $a \in \Sigma$ and states $q \in Q_{\mathcal{PD}}$. The set of final states is defined as $F = \{q \in Q_{\mathcal{PD}} \,|\, \varepsilon(q) = \varepsilon\}$.

Recalling from 2.3.3.1, the linear form can be used as an efficient way to compute partial derivatives. Let $\alpha \in \mathit{RegExp}$ over Σ and $a \in \Sigma$:

$$\delta(\alpha, a) = \partial_a(\alpha) = \{\beta \mid (a, \beta) \in lf(\alpha)\}.$$

That is, to compute the set of partial derivatives of α with respect to a symbol a, one inspects the linear form of α and collects all continuations β that follow an a in the prefix position. This characterization is not only concise but also algorithmically useful: it gives a direct recipe for building the transition relation of the partial derivatives automaton.

Theorem 2.4.1 ([1])**.** For every regular expression α, the partial derivatives automaton $\mathcal{N}_\alpha$ accepts exactly the language $L(\alpha)$. Moreover, the number of states of $\mathcal{N}_\alpha$ is bounded by $|\alpha| + 1$.

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. [2]

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 [4] 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 Finite Automata

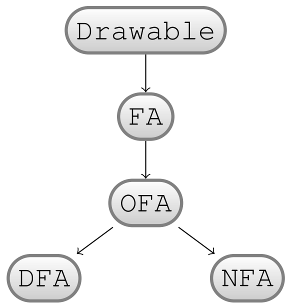

Figure 2.5: Class inheritance organization in FAdo's finite automata code file, `fa.py` (excluding the `SemiDFA` class)

As shown in Figure 2.5, FAdo organizes its implementation of finite automata as an inheritance hierarchy. At the top level, the class `Drawable` provides only drawing functionality, relying on the `graphviz` backend to visualize automata.

On top of this, the class `FA` defines the abstract interface for finite automata, introducing the fundamental components: the set of states, the alphabet, the transition function, the initial state(s), and the set of final states. This class also establishes common methods for manipulating these elements and for interoperability between automata.

The class `OFA` ("one-way finite automaton") specializes `FA` and serves as the immediate superclass for concrete models. It defines abstract operations such as evaluation of symbols, addition of transitions, and access to useful states, ensuring a consistent contract across different automaton types.

Two subclasses will then inherit from `OFA`:

- `NFA` — implements nondeterministic finite automata, possibly with ε-transitions. It provides methods for epsilon-closure, elimination of ε-moves, subset construction into a DFA, and regular operations such as union, concatenation, and Kleene star.

- `DFA` — implements deterministic finite automata. It enforces determinism by construction and provides efficient word evaluation and minimization routines.

2.5.2 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.

Chapter 3

State of the Art

3.1 Introduction

Regular expressions (regex) remain one of the most powerful and widely adopted tools for string pattern matching across programming languages, search tools, and data processing pipelines. While their theoretical foundation lies in formal language theory, the practical implementation of regex engines often diverges from the idealized models. This chapter mentions the state of the art in regex engine designs and regex language features, with a particular focus on performance trade-offs, security concerns, and evolving capabilities.

3.2 Engine Architectures

Regex engines are typically implemented using one of the following architectures:

3.2.1 Deterministic Finite Automata (DFA)

DFA-based engines compile a regular expression into a finite-state machine that reads each input character exactly once, guaranteeing linear-time performance.

3.2.2 Backtracking Engines

Backtracking engines simulate nondeterministic finite automata (NFA) and recursively explore different matching paths. They are expressive but can suffer from exponential worst-case behavior.

For example, .NET's regex engine uses a traditional nondeterministic finite automaton that is also used in Perl, Python, Emacs and Tcl.

3.2.3 Hybrid Engines

Modern engines often combine DFA and backtracking strategies. For example, they may use a DFA to fast-forward through non-ambiguous parts and switch to backtracking only when needed.

3.3 Feature Sets of Regex Languages

Regex languages have evolved far beyond classical regular expressions as defined in formal language theory. The following extensions are now standard in most industrial-strength engines:

3.3.1 Backreferences

Backreferences allow the engine to refer to previously captured groups. This enables matching non-regular patterns (e.g., repeated substrings) but breaks the regular language model.

3.3.2 Lookahead and Lookbehind

These zero-width assertions check what follows or precedes a pattern without consuming input. They are useful for complex validations but add significant complexity.

3.3.3 Unicode and Multilingual Support

Modern engines increasingly support Unicode properties (e.g., `\p{L}` for letters) and normalization, essential for multilingual applications.

3.3.4 Named Capture Groups and Subroutines

Named groups (`(?<name>...)`) and recursive subpatterns (e.g., `(?R)`) have become essential for advanced pattern extraction.

3.3.5 Flags and Modes

Regex engines support flags for case insensitivity, multiline matching, dot-all mode (where `.` matches newlines), and others.

3.4 Engines and Libraries

3.4.1 RE2

Developed by Google, RE2 is a DFA-based engine designed to never exceed linear time or memory. It disallows features like backreferences for safety and predictability.

3.4.2 PCRE2

An industry-standard library used in PHP and many scripting tools. It supports extensive features including backtracking control verbs and recursive patterns.

3.4.3 Hyperscan

Hyperscan is Intel's regular expression matching engine, designed specifically for high-throughput and low-latency applications. It serves as a core component in several security and networking tools, including intrusion detection systems like Suricata and firewalls.

Traditional regex engines (e.g., backtracking-based ones like PCRE) often struggle with performance bottlenecks due to their sequential nature and vulnerability to ReDoS attacks. Hyperscan addresses these limitations by combining multiple automata models—particularly NFAs and DFAs—with a hybrid execution strategy that leverages Single Instruction, Multiple Data (SIMD) parallelism and tiled execution on modern CPUs .

According to Wang et al. ([7]), Hyperscan divides regexes into multiple subgraphs, such as anchored DFAs for simple patterns and NFAs for complex constructs. This hybrid approach enables it to process large volumes of data streams efficiently without the exponential-time risks associated with backtracking engines.

However, as also noted in [7], Hyperscan will also enforce syntactic restrictions when compiling. Regexes that are deemed vulnerable or ambiguous are rejected, therefore limiting expressiveness and versatility, especially when the user is looking for nested repetition (e.g. $(a+)+$, looking to match one or more of one or more a's) or greedy alternation (e.g. $(a|aa)+$, matching a sequence of a or aa, repeated, resulting in three matches for a string such as aa).

3.4.4 Rust's `regex` Crate

Implements a hybrid DFA/NFA model and guarantees linear-time performance by excluding features like backreferences.

3.5 Prevalency of ReDoS

3.5.1 Revealer

Mention the revealer paper here!

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. Current tools are already able to do this, including some non-backtracking matchers.

4.2 Derivatives of operations in Extended Regular Expressions

Given a word $w = xyz$ and a regular expression α, $w \in L(\alpha)$ if $\varepsilon(D_w(\alpha)) = \varepsilon$. In order to match with fixed and bounded repetition, one must first solve the derivatives for those operations.

4.2.1 Fixed Repetition

For a fixed repetition, we need r to occur n times, resulting in the notion r^n.

Given $r = ab$ such that $r^2 = abab$, we have:

$$\begin{aligned} D_a(r^2) &= D_a(r \cdot r) \\ &= D_a(abab) \\ &= D_a(a)bab + D_a(bab) \\ &= \varepsilon bab + \emptyset \\ &= bab \end{aligned}$$

4.2.2 Bounded Repetition

Supplementing the fixed repetition notion, we now add a maximum bound so that r can occur at least n times and at maximum $m-1$ times: $r^{[n,m[}$.

Given $r = ab$ such that $r^2 = abab$ and $r^3 = ababab$, we have:

$$\begin{aligned} D_a(r^{[2,4[}) &= D_a(abab) + D_a(ababab) \\ &= D_a(abab) + D_a(ababab) \\ &= D_a(a)bab + D_a(bab) + D_a(a)babab + D_a(babab) \\ &= \varepsilon bab + \emptyset + babab + \emptyset \\ &= bab + babab \end{aligned}$$

This also extends for $r^{[m,\mathrm{inf}[}$, which is the same as having $r^m \cdot r^*$:

$$\begin{aligned} D_a(r^{[2,\mathrm{inf}[}) &= D_a(abab) + D_a((ab)^*) \\ &= D_a(a)bab + D_a(bab) + D_a(ab) \cdot (ab)^* \\ &= \varepsilon bab + \emptyset + (D_a(a)b + D_a(b)) \cdot (ab)^* \\ &= bab + (\varepsilon b + \emptyset) \cdot (ab)^* \\ &= bab + b(ab)^* \end{aligned}$$

The regular expression $r^{[2,\mathrm{inf}[}$ will match any $(ab)^k$ with $k \geq 2$.

4.3 Class implementation in FAdo

In order to implement these new syntactic constructs of extended regular expressions in FAdo, we added exact-power and counted-repetition nodes to the regular expression abstract syntax tree and define their derivatives, partial derivatives and linear form cases.

4.3.1 CPower

CPower is the class responsible for the construction of regular expressions using the fixed repetition operator.

```
class CPower(Unary):
   def __init__(self, arg, n, sigma=None):
   self.arg = arg
   self.n = n
   self.Sigma = sigma
   self._ewp = False if self.n > 0 else True

   ...
```

As shown above, the class inherits from the *Unary*, which only defines the *Unary.Sigma* and *Unary.arg* properties, letting the class hold a alphabet set (representing Σ) and the symbol used to construct this operator.

In order to integrate the class fully with *FAdo*, some more methods had to be implemented.

```
   def linearForm(self):
      arg_lf = self.arg.linearForm()
      lf = dict()
      for head in arg_lf:
         lf[head] = set()
         for tail in arg_lf[head]:
            if self.n == 0:
               lf[head].add(CEmptySet(self.Sigma))
            elif self.n == 1:
               lf[head].add(CEpsilon(self.Sigma))
            else:
               lf[head].add(CPower(self.arg, self.n-1, self.Sigma))

      return lf

   def partialDerivatives(self, sigma):
      return self.arg.partialDerivatives(sigma)

   def derivative(self, sigma):
      if str(sigma) in str(self.arg):
         if self.n == 0:
            return CEmptySet(sigma)
         elif self.n == 1:
            return self.arg.derivative(sigma)
         else:
            return CConcat(self.arg.derivative(sigma), CPower(self.arg,
                self.n-1, self.Sigma))
      else:
         return CEmptySet(sigma)
```

4.4 Extended Regular Expressions in FAdo

Recalling back from 2.3.1, in order to support extended regular expressions, one needs to extend the base class for unary operations in FAdo

Chapter 5

Matching

Matching is the process of checking whether a piece of text fits a specific pattern described using a regular expression (regex). 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 a modified 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 Overlapped versus Non-Overlapped Matching

In the context of regular expression matching, two distinct paradigms exist: *overlapped matching* and *non-overlapped matching*. Understanding their differences is crucial when designing matching engines, especially when completeness or performance is a concern.

Overlapped Matching

Overlapped matching refers to finding all possible matches of a pattern in an input string. This is typically achieved by attempting a match starting at every index of the input. It is more exhaustive and useful in domains where no potential match should be missed, such as bioinformatics (DNA pattern searching).

For example, the indexed string $w = a_0a_1a_2a_3$, when matched against using the pattern aa, will yield the following matched substrings:

- $w_{[0,2[} = aaaa$
- $w_{[1,3[} = aaaa$
- $w_{[2,4[} = aaaa$

Non-Overlapped Matching

Non-overlapped matching (also referred to as *disjoint*, *standard*, or in some engines, *greedy* matching) finds matches sequentially from left to right, and once a match is found, it advances the input pointer beyond the match. Unlike overlapped matchers, where overlapping is a feature by default, greedy matchers will often depend on the lookahead assertions to do so.

Using the same pattern aa on $w = a_0a_1a_2a_3$, a non-overlapped matcher may return:

- $w_{[0,2[} = aaaa$
- $w_{[2,4[} = aaaa$

To summarize, overlapped matching provides a more complete view of potential matches, but at a higher computational cost. It is particularly well-suited to automata-based approaches like the modified position automaton described in this work.

5.2 Modified Position Automata

A *position automaton* is a type of nondeterministic finite automaton (NFA). We can enable overlapped matching by modifying the position automaton's construction using the algorithm described on 1.

Algorithm 1 NFAPOSCOUNT(R): Construct Special Position Automaton

Require: Regular expression R
Ensure: NFA A

```
A ← new empty NFA
i ← A.addInitialState()
A.addTransitionStar(i, i)                    ▷ Accept any symbol from Σ
f_R ← R.marked()
stack ← empty stack
addedStates ← empty map
for all p ∈ First(f_R) do
    q ← A.addState(p)
    addedStates[p] ← q
    stack.push((p, q))
    A.addTransition(i, p, q)
end for
while stack is not empty do
    (s, s_idx) ← stack.pop()
    for all t ∈ Follow(f_R, s) do
        if t ∈ addedStates then
            q ← addedStates[t]
        else
            q ← A.addState(t)
            addedStates[t] ← q
            stack.push((t, q))
        end if
        A.addTransition(s_idx, t, q)
    end for
end while
e ← A.addState()
A.addTransitionStar(e, e)
for all p ∈ f_R.Last() do
    if p ∈ addedStates then
        A.addFinal(addedStates[p])
        A.addTransitionStar(addedStates[p], e)
    end if
end for
```

5.3 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+)+$
```

5.4 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

```
symbols ← list with ε prepended to s
currentRow ← empty map from states to list of position pairs
finalMatches ← empty map from states to list of matches
position ← 0
for all sym in symbols do
    if sym = ε then
        for all q_0 ∈ I do
            currentRow[q_0] ← [(0, 0)]
        end for
    else
        nextRow ← empty map
        if sym ∈ Σ then
            for all q ∈ keys(currentRow) do
                if |δ(q, sym)| > 0 then
                    for all q' ∈ δ(q, sym) do
                        for all (start, _) ∈ currentRow[q] do
                            if q' = q and q' ∈ I then
                                append (position, position) to nextRow[q']
                            else
                                append (start, position) to nextRow[q']
                                if q' ∈ F then
                                    append (start, position) to finalMatches[q']
                                end if
                            end if
                        end for
                    end for
                end if
            end for
        else                                          ▷ Symbol not in Σ; treat as fresh start
            for all q_0 ∈ I do
                nextRow[q_0] ← [(position, position)]
            end for
        end if
        currentRow ← nextRow
        position ← position + 1
    end if
end for
return finalMatches
```

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

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

We can separate R into two matching groups:

- The first group $(aa + aaa)$ will match either two or three a symbols (e.g. aaa will yield three overlapped matches: aaa, aaa and aaa).
- The second group $(aaa + aa)$ will also match either two or three a symbols, much like the first group.

The Glushkov automaton construction for R is as follows:

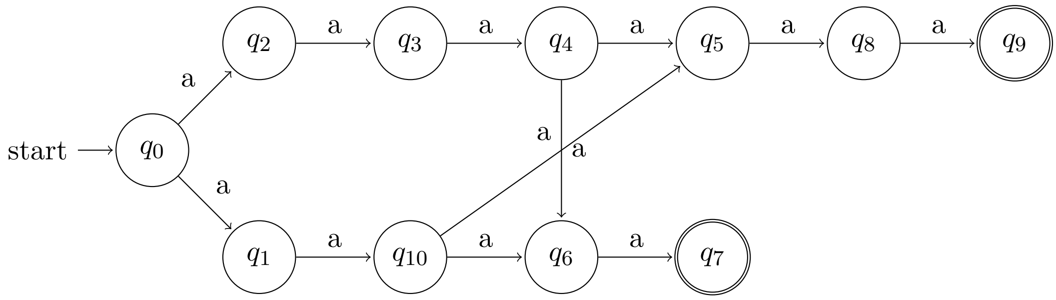

Figure 5.1: Default Glushkov automaton

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

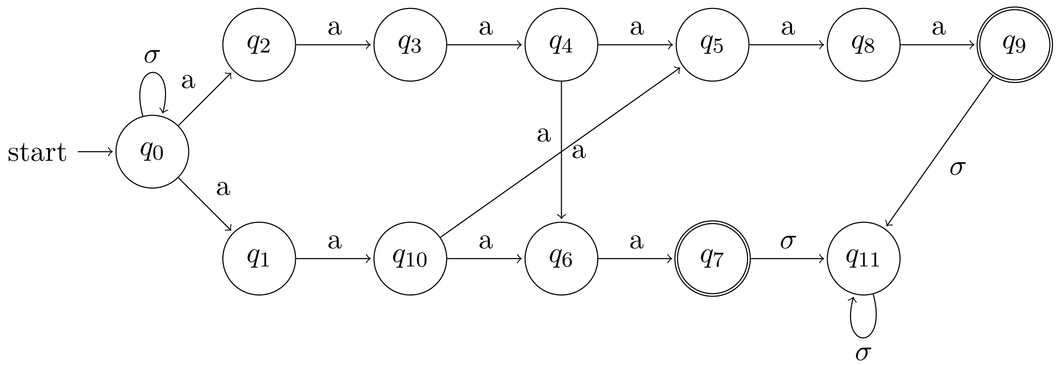

Figure 5.2: Modified Glushkov automaton

When we apply the matching algorithm (2) to an input string, it will traverse the automaton and record all positions where matches occur. This approach ensures that we can find all possible matches, including those that overlap, without falling into the exponential blowup trap of backtracking.

The result is a table that maps each accepting state to a set of index pairs, each indicating the start and end of a successful match. For instance, applying this process to $R = (aa+aaa)(aaa+aa)$ and $w = aaaaabaaaaa$ will yield the following table:

Table 5.1: Match positions table using the regular expression $R = (aa + aaa)(aaa + aa)$ and input string $w = aaaaabaaaaa$

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
ε	(0,0)											
a	(1,1)	(0,1)	(0,1)									
a	(2,2)	(1,2)	(1,2)	(0,2)							(0,2)	
a	(3,3)	(2,3)	(2,3)	(1,3)	(0,3)	(0,3)	(0,3)				(1,3)	
a	(4,4)	(3,4)	(3,4)	(2,4)	(1,4)	(0,4) (1,4)	(0,4) (1,4)	(0,4)	(0,4)		(2,4)	
a	(5,5)	(4,5)	(4,5)	(3,5)	(2,5)	(1,5) (2,5)	(1,5) (2,5)	(1,5) (0,5)	(1,5) (0,5)	(0,5)	(3,5)	(0,5)
b	(6,6)											
a	(7,7)	(6,7)	(6,7)									
a	(8,8)	(7,8)	(7,8)	(6,8)							(6,8)	
a	(9,9)	(8,9)	(8,9)	(7,9)	(6,9)	(6,9)	(6,9)				(7,9)	
a	(10,10)	(9,10)	(9,10)	(8,10)	(7,10)	(6,10) (7,10)	(6,10) (7,10)	(6,10)	(6,10)		(8,10)	
a	(11,11)	(10,11)	(10,11)	(9,11)	(8,11)	(7,11) (8,11)	(7,11) (8,11)	(6,11) (7,11)	(6,11) (7,11)	(6,11)	(9,11)	(6,11)

First, we always have to account for ε, since there is the possibility of having the empty word and we also want to match against it. After that, every symbol $s \in w$ is processed sequentially.

At each step, the algorithm updates a row that maps the automaton's (represented in 5.2) states to sets of position intervals (i, j), such that the substring w_{ij} corresponds to a valid match, whether it is overlapped or not.

Transitions are computed for each input symbol using the automaton's δ function. When a final state is reached, the interval is stored as a successful match. Furthermore, during this process, only the last symbol's computed transitions and position intervals are preserved because they always carry over to the current symbol's computation.

The automaton on 5.2 shows that:

$$F = \{q_7, q_9\}$$

For those states, the resulting match table yields the following:

Table 5.2: Highlighted substrings of valid matches for state q_7

$w_{0,4}$	aaaaabaaaaa
$w_{0,5}$	aaaaabaaaaa
$w_{1,5}$	aaaaabaaaaa
$w_{6,10}$	aaaaabaaaaa
$w_{6,11}$	aaaaabaaaaa
$w_{7,11}$	aaaaabaaaaa

Table 5.3: Highlighted substrings of valid matches for state q_9

$w_{0,5}$	aaaaabaaaaa
$w_{6,11}$	aaaaabaaaaa

Recalling the regular expression used $R = (aa + aaa)(aaa + aa)$ and input string $w = aaaaabaaaaa$,

Bibliography

[1] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996. ISSN: 0304-3975. doi:https://doi.org/10.1016/0304-3975(95)00182-4.

[2] 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.

[3] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964. ISSN: 0004-5411. doi:10.1145/321239.321249.

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

[5] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. ISSN: 0001-0782. doi:10.1145/363347.363387.

[6] Ian Erik Varatalu, Margus Veanes, and Juhan Ernits. Re#: High performance derivative-based regex matching with intersection, complement, and restricted lookarounds. *Proceedings of the ACM on Programming Languages*, 9(POPL):1–32, January 2025. ISSN: 2475-1421. doi:10.1145/3704837.

[7] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 631–648, Boston, MA, February 2019. USENIX Association. ISBN: 978-1-931971-49-2.

[8] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 631–648, Boston, MA, February 2019. USENIX Association. ISBN: 978-1-931971-49-2.

[9] Isaac Z. Schlueter. minimatch, 2011.