

Daniil Belyakov


A FURTHER DEVELOPMENT OF A
GENERIC JAVASCRIPT-BASED LAN-
GUAGE MATCHING FRAMEWORK

Bachelor's Thesis
in Information Technology

November 2015



DESCRIPTION

| | | |
|--|--|--|
|  | | Date of the bachelor's thesis 27.11.2015 |
| Author(s) Daniil Belyakov | Degree programme and option Information Technology | |
| Name of the bachelor's thesis A Further Development of a Generic JavaScript-Based Language Matching Framework | | |
| Abstract Whynot is a generic JavaScript-based language matching framework, built by a lead developer of Dutch web development company Liones. According to the developer's assumption, change of the underlying structure of the framework from Nondeterministic Finite Automata to Deterministic Finite Automata could be profitable for the performance of the framework. This thesis tests the assumption: it covers the theoretical topics of Finite Automata, Regular Languages and Algorithm Complexity. The theory is further used as a base for building a simplified version of Whynot with Deterministic Finite Automata as underlying structure and testing it against the original Whynot. The test results show that, under a certain assumption, the change of the underlying structure of the framework to Deterministic Finite Automata leads to decreased system complexity. Moreover, the new system is found useful as a base for further researches of possible Whynot developments. | | |
| Subject headings, (keywords) Algorithm Complexity, Finite Automata, JavaScript, Language Matching, Regular Languages, Whynot.js | | |
| Pages 55 | Language English | URN |
| Remarks, notes on appendices | | |
| Tutor Jari Kortelainen | Employer of the bachelor's thesis Mikkeli University of Applied Sciences | |

CONTENTS

| | | |
|-----|---|----|
| 1 | INTRODUCTION..... | 1 |
| 2 | FINITE AUTOMATA..... | 3 |
| 2.1 | Overview..... | 3 |
| 2.2 | Deterministic Finite Automata..... | 5 |
| 2.3 | Nondeterministic Finite Automata..... | 6 |
| 2.4 | Nondeterministic Finite Automata with ϵ -transitions..... | 8 |
| 2.5 | Converting Nondeterministic Finite Automata to Deterministic Finite Automata..... | 9 |
| 2.6 | DFA Minimization..... | 13 |
| 3 | REGULAR EXPRESSIONS..... | 17 |
| 3.1 | Overview..... | 17 |
| 3.2 | Implementation..... | 19 |
| 4 | COMPLEXITY OF ALGORITHMS..... | 25 |
| 4.1 | Overview..... | 25 |
| 4.2 | Algorithm Complexity Metrics..... | 26 |
| 4.3 | Algorithm Complexity Analysis..... | 28 |
| 5 | A GENERIC, JAVASCRIPT-BASED FORMAL LANGUAGE MATCHING FRAMEWORK, WHYNOT.JS..... | 31 |
| 5.1 | Overview..... | 31 |
| 5.2 | Structure and Operation..... | 33 |
| 5.3 | Weak Spots and Further Development..... | 33 |
| 6 | COMPILING THE MINIMAL DETERMINISTIC FINITE AUTOMATA..... | 35 |

| | | |
|----------|---|-----------|
| 6.1 | Overview | 35 |
| 6.2 | Development and Implementation | 36 |
| 6.3 | Suggested Further Development | 42 |
| 7 | COMPLETING AND EXTENDING THE INPUT | 43 |
| 7.1 | Overview | 43 |
| 7.2 | Development and Implementation | 43 |
| 7.3 | Systems Comparison..... | 50 |
| 7.4 | Further Development..... | 51 |
| 8 | CONCLUSIONS | 52 |
| | BIBLIOGRAPHY | 53 |

1 INTRODUCTION

The modern world is undoubtedly based on information. Information is present everywhere in exponentially growing amounts, and it needs to be managed efficiently. The task of information management is especially significant for the companies working with large documentations. Among the technologies used for such documentations management is XML (World Wide Web Consortium 2015).

DEFINITION 1. (World Wide Web Consortium 2015) *Extensible Markup Language*, or XML is a human- and machine- readable text format to store large-scale electronic publications and described as a set of free and open standards.

XML provides a way to store data along with its context omitting any representational details. XML documents consist of *tags*, describing the data context, their attributes, storing the properties of tags, and their content, representing the data itself. XML allows for creation of custom tags and, thus, serves as a base for narrower, application-specific formats. Custom formats are defined by custom XML *schemas* — files, describing the tags of the formats, their attributes and inter-relations between them, such as tags nesting. Schema rules are strict constraints, and their violation is supposed, according to the XML standard, to fail the erroneous document processing in a hard manner. (Silmaril Consultants 2015)

Traditionally, creation and maintenance of XML documentations was requiring two separate departments: authors, responsible for writing the documentation itself and XML specialists, putting the texts written by authors into the proper format XML document. However, Dutch company *Liones* has developed a solution eliminating the need in most of the time- and resource-consuming job done by XML departments: a user-friendly web-based XML editor, requiring no XML knowledge (Liones 2015). The editor is called *FontoXML* or just *Fonto*, and it allows users to edit XML documents as if they were working with conventional text editors, such as Microsoft Word ® (FontoXML 2015).

Fonto is targeted at user friendliness (FontoXML 2015). Thus, its smooth operation is vital and, therefore, optimization of the heavily loaded components of Fonto is a primal task. Since Fonto allows editing XML in real time, it relies on the real time input validation, completion and extension system. For every input a user introduces, the system is intended to answer the following questions:

- Is the input valid in respect with a given schema?
- If the input is valid, what are the possible useful, in terms of the system, extensions of the input saving its validity?
- If the input is invalid, is it possible to complete the input to make it valid? If it is, then what are the possible completions?

This thesis suggests a further development of the above-described system. The system is based on nondeterministic finite automata computational model and the model is currently assumed to be a weak spot of the system. Thus, the research questions of the thesis are as follows:

- What are finite automata?
- What are regular expressions?
- How are finite automata related to regular expressions?
- How to measure complexity of a system components (algorithms)?
- How would change of the system underlying computational model from nondeterministic to deterministic finite automata affect the system complexity and performance?

Chapters 2 to 4 provide an overview of finite automata, regular expressions and algorithmic complexity, Chapter 5 describes the original system and proposes the development, Chapters 6 to 7 report the implementation process of the development in detail, and finally, Chapter 8 is devoted to conclusions.

2 FINITE AUTOMATA

2.1 Overview

A *Finite Automaton* (FA) [plural: *automata*, from Latin] is a computational model used to *accept* or *reject* strings of symbols. An FA consists of a finite set of states, a subset of which is marked as *initial* and a subset — as *accepting*, a set of possible input symbols, referred to as *alphabet* of the automaton, and a set of *transitions* between the states, bound to the elements of the alphabet (Hopcroft et al. 2003). Generally speaking, an FA can be in one or more states at a time.

Finite automata execution principle is as follows: an FA takes symbols of an input string one by one and, for every of its current states, either traverses a transition bound to the symbol, or ends its execution in the state either accepting or rejecting the input string. The input string is considered to be accepted if the FA reaches the end of input having at least one accepting state among its current states. If, on the other hand, either the FA fails in all of its current states simultaneously, or the end of input is reached with none of the current states being accepting, the string is said to be rejected (Hopcroft et al. 2003). An important consequence of the FA operation principle is that FA execution does not depend on its execution history, such as previously visited states or past input (Cormen et al. 2001).

There is a multitude of ways to depict an FA, but the most demonstrative one is a directed graph (Hopcroft et al. 2003; Brookshear 1989). We will use this fashion of FA depiction throughout this thesis.

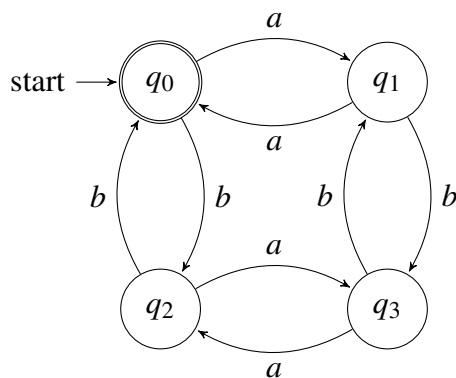


FIGURE 1. A FA as a directed graph

EXAMPLE 1. The diagram in Figure 1 represents a directed graph corresponding to an FA which accepts the strings containing only a 's and/or b 's, each in even amounts, and an empty string. The FA elements are represented in the following way:

- Circles are states.
- Arrows are transitions.
- Symbols near the arrows are inputs triggering the transitions.
- Arrow originating from the "start" label points to the initial state.
- Double circle(s) are accepting state(s).

Finite Automata are widely used to describe various physical and logical structures in both everyday life and science. One casual example is a vending machine coin accepting system. The most notorious scientific use, on the other hand, is as means of lexical analysis, where Finite Automata are used to accept or reject sets of strings. Important is the fact that any given problem describable by an FA is also representable as a lexical analysis problem. (Brookshear 1989)

A Finite Automaton can either be deterministic (DFA) or nondeterministic (NFA). The difference between the two is that while DFA can only be in exactly one state at a time, NFA can be in one or more possible states simultaneously (Hopcroft et al. 2003). There are other, significantly more sophisticated approaches to nondeterminism, which are, however, out of scope of this thesis. Consequently, the above approach will be used throughout the thesis.

2.2 Deterministic Finite Automata

A *Deterministic Finite Automaton* (DFA), is an FA which can only have a single transition for every input symbol from any given state to another, or, from the other perspective, can only be in a single state at a time (Hopcroft et al. 2003).

DEFINITION 2. (Hopcroft et al. 2003; Brookshear 1989) DFA is a five-tuple $A = (Q, \Sigma, \delta, \iota, F)$, where:

1. Q is a finite set of states.
2. Σ is an alphabet of A .
3. δ is a function from $Q \times \Sigma$ into Q , the transition function of A .
4. ι is an initial state of A .
5. F is a set of accept states of A .

The set of strings accepted by an FA is referred to as the language L of the FA (Hopcroft et al. 2003). Thus, FA are means of describing regular languages — a subset of formal languages.

DEFINITION 3. (Reghizzi 2009) A *formal language* L over an alphabet Σ is a subset of Σ^* , the set of all the words over Σ , constrained using formal rules.

In the above definition, Σ^* is a set consisting of an empty string and all the concatenations of characters contained in Σ with lengths of one or more (Hopcroft et al. 2003).

DEFINITION 4. (Brookshear 1989) A *regular language* is a language accepted by an FA.

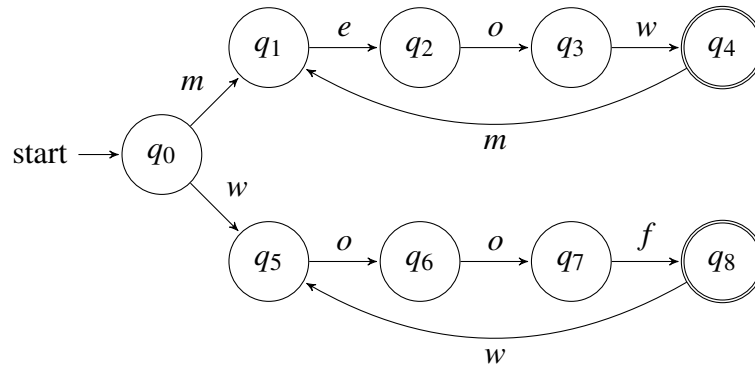


FIGURE 2. DFA example

EXAMPLE 2. A diagram of a DFA accepting strings of either "meow" or "woof" patterns of positive length is depicted in Figure 2. The automaton components are as follows:

1. $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\}$
2. $\Sigma = \{e, f, m, o, w\}$
3. δ is defined in the following way: $\delta(q_0, m) = q_1$, $\delta(q_0, w) = q_5$, $\delta(q_1, e) = q_2$, $\delta(q_2, o) = q_3$, $\delta(q_3, w) = q_4$, $\delta(q_4, m) = q_1$, $\delta(q_5, o) = q_6$, $\delta(q_6, o) = q_7$, $\delta(q_7, f) = q_8$, $\delta(q_8, w) = q_5$
4. $\iota = q_0$
5. $F = \{q_4, q_8\}$

The following holds for the automaton:

- $\{\text{"meow"}, \text{"woof"}, \text{"meowmeow"}, \text{"woofwoof"}, \text{"meowmeowmeow"}, \dots\}$ is the language L of the automaton.
- "meow", "woof", "meowmeow" etc. are examples of particular words accepted by the automaton.
- Automaton itself describes a rule constraining the language L .

2.3 Nondeterministic Finite Automata

A *Nondeterministic Finite Automaton* (NFA), is an FA which can have one or more transitions from a state to another for each possible input symbol (Hopcroft et al. 2003). From the other perspective, an NFA could be in more than one state simultaneously (Brookshear

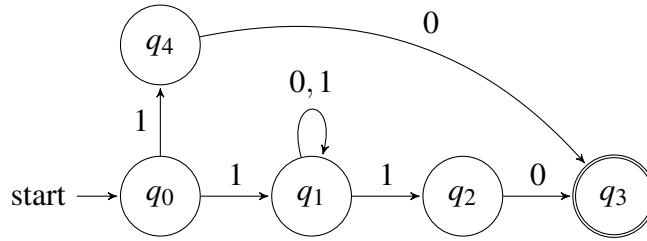


FIGURE 3. NFA Example

1989).

DEFINITION 5. (Hopcroft et al. 2003; Brookshear 1989) NFA is a 5-tuple $A = (Q, \Sigma, \delta, \iota, F)$, where:

1. Q is a finite set of states.
2. Σ is an alphabet of A .
3. $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$, where $\mathcal{P}(Q)$ is the powerset of Q , is the transition function of A .
4. ι is a set of initial states of A .
5. F is a set of accept states of A .

EXAMPLE 3. A diagram of an NFA accepting all the even binary numbers not divisible by binary number 100 is depicted in Figure 3. The automaton formal components are as follows:

1. $Q = \{q_0, q_1, q_2, q_3, q_4\}$
2. $\Sigma = \{0, 1\}$
3. δ is defined in the following way: $\delta(q_0, 1) = \{q_1, q_4\}$, $\delta(q_1, 0) = \{q_1\}$, $\delta(q_1, 1) = \{q_1, q_2\}$, $\delta(q_2, 0) = \{q_3\}$, $\delta(q_4, 0) = \{q_3\}$
4. $\iota = \{q_0\}$
5. $F = \{q_3\}$

Although NFA definitely provide larger degree of convenience in languages representation compared to DFA, they are not the most laconic finite automata type (Brookshear 1989). The following section tells about the finite automata type allowing to describe an

underlying language with the least representational effort possible among FA.

2.4 Nondeterministic Finite Automata with ε -transitions

DEFINITION 6. (Hopcroft et al. 2003) An ε -transition is a transition which requires no input symbol.

A convenient expansion to NFA is allowance of ε -transitions. These transitions let automata change their states without taking any input. An NFA allowing the use of ε -transitions is called ε -NFA, and the ε itself is referred to as an *empty character* (Hopcroft et al. 2003).

DEFINITION 7. (Hopcroft et al. 2003) ε -NFA is a 5-tuple $E = (Q, \Sigma^+, \delta, \iota, F)$, where:

1. Q is a finite set of states.
2. $\Sigma^+ = \Sigma \cup \{\varepsilon\}$ is an extended alphabet of E .
3. $\delta : Q \times \Sigma^+ \rightarrow \mathcal{P}(Q)$, where $\mathcal{P}(Q)$ is the power set of Q , is the transition function of E .
4. ι is a set of initial states of E .
5. F is a set of accept states of E .

From Definition 7 and Definition 5 it is clear that $\Sigma \subseteq \Sigma^+$; thus, every NFA is also an ε -NFA with exactly zero ε -transitions.

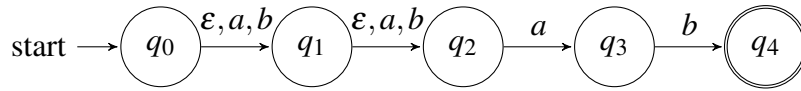


FIGURE 4. NFA with ε -transitions example

EXAMPLE 4. Figure 4 depicts an ε -NFA accepting the following language: $\{ "ab", "aab", "bab", "aaab", "abab", "baab", "bbab" \}$. The automaton formal components are as follows:

1. $Q = \{q_0, q_1, q_2, q_3, q_4\}$
2. $\Sigma^+ = \{a, b, \varepsilon\}$
3. δ is defined in the following way: $\delta(q_0, \varepsilon) = \{q_1\}$, $\delta(q_0, a) = \{q_1\}$, $\delta(q_0, b) = \{q_1\}$, $\delta(q_1, \varepsilon) = \{q_2\}$, $\delta(q_1, a) = \{q_2\}$, $\delta(q_1, b) = \{q_2\}$, $\delta(q_2, a) = \{q_3\}$, $\delta(q_3, b) = \{q_4\}$
4. $\iota = \{q_0\}$
5. $F = \{q_4\}$

As Hopcroft et al. state, among all the finite automata, ε -NFA are the most convenient and widely used for Lexical Analysis (Hopcroft et al. 2003). However, according to Brookshear, DFA are significantly easier to simulate (Brookshear 1989). Thus, the following section presents a way to convert ε -NFA to DFA.

2.5 Converting Nondeterministic Finite Automata to Deterministic Finite Automata

THEOREM 1. (Hopcroft et al. 2003) *For every ε -NFA there is a DFA accepting exactly the same language.*

Whereas proof of the Theorem 1 is beyond the scope of the thesis, the fact it proves is important.

In their article, Rabin & Scott present an algorithm for NFA to DFA conversion, called *Powerset Construction* (Rabin & Scott 1959). For a given NFA, the algorithm constructs a DFA, in which each state corresponds to a subset of the original NFA states set.

In a later work, Schneider introduces a further development of the algorithm by Rabin & Scott, allowing for ε -NFA to DFA conversion only considering the states accessible from the original automaton initial states set (Schneider 2003). The algorithm will be referenced to as $\text{DETERMINIZE}(E)$ throughout this thesis.

DEFINITION 8. (Hopcroft et al. 2003) An ε -closure of a state q of an ε -NFA is a set consisting of q and all the states accessible from it by a chain of one or more ε -transitions.

The $\text{DETERMINIZE}(E)$ algorithm is based on the concept of ε -closure. Thus, its implementation requires a sub-algorithm to find ε -closures. Such an algorithm is presented in the book of Hopcroft et al. under the name $\text{ECLOSE}(q)$, where q is a given state (Hopcroft et al. 2003). For convenience, the version presented in this thesis has a reference to the target ε -NFA added as a first argument: $\text{ECLOSE}(E, q)$.

Since, as mentioned in Section 2.4, every NFA is an ε -NFA, $\text{DETERMINIZE}(E)$ is equally suitable for both ε -NFA and NFA without any ε -transitions.

ALGORITHM 1. Finding epsilon closure of a state

Require: $E = (Q_E, \Sigma^+, \delta_E, \iota_E, F_E)$ is an ε -NFA, q is a state in Q_E

```

1: function  $\text{ECLOSE}(E, q)$ 
2:    $\text{closure} \leftarrow \{q\}$  ▷ Initialize array to store  $\varepsilon$ -closure of  $q$ 
3:    $\text{neighbors} \leftarrow \delta_E(q, \varepsilon)$  ▷ Get the set of neighbors — states accessible from  $q$  through a single  $\varepsilon$ -transition
4:   if  $\text{neighbors}$  is not empty then
5:     for each state in  $\text{neighbors}$  do
6:        $\text{closure} \leftarrow \text{closure} \cup \text{ECLOSE}(E, \text{state})$  ▷ Recursively add  $\varepsilon$ -closure of every neighbor to the  $\varepsilon$ -closure of  $q$ 
7:   return  $\text{closure}$ 

```

Ensure: closure is an ε -closure of q

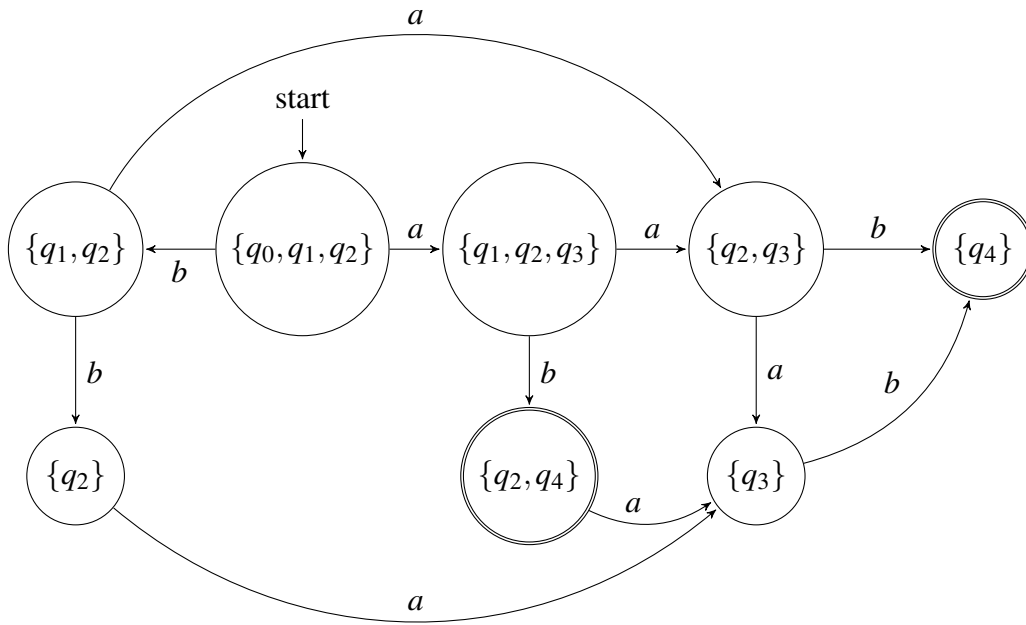


FIGURE 5. A DFA converted from the ε -NFA depicted in Figure 4 Using Algorithm 2

EXAMPLE 5. Calling ECLOSE function with the automaton depicted in Figure 4 as a first argument and q_0 as a second would return $\{q_0, q_1, q_2\}$, since:

- q_0 is the state passed to the function.
- q_1 is directly accessible from q_0 through an ε -transition.
- q_2 is directly accessible from q_1 through an ε -transition, and, thus, accessible from q_0 through a chain of ε -transitions.

EXAMPLE 6. Calling DETERMINIZE(E) function with ε -NFA depicted in Figure 4 as an argument would return the DFA depicted in Figure 5.

ALGORITHM 2. Powerset construction with ε -transitions removal by Schneider

Require: $E = (Q_E, \Sigma^+, \delta_E, \iota_E, F_E)$ is an ε -NFA

```

1: function DETERMINIZE( $E$ )
2:    $D \leftarrow (Q_D, \Sigma, \delta_D, \iota_D, F_D)$  ▷ Initialize a DFA to store
   the result
3:    $Q_{DL} \leftarrow []$ 
4:    $Q_{DL}[0] \leftarrow \{\}$ 
5:   for each initialState  $\in \iota_E$  do ▷ Find initial state of the
   new DFA
6:      $Q_{DL}[0] \leftarrow Q_{DL}[0] \cup \text{ECLOSE}(E, \text{initialState})$ 
7:    $\iota_D \leftarrow Q_{DL}[0]$ 
8:    $n \leftarrow 0$ 
9:   while  $n < |Q_{DL}|$  do ▷ As long as there are more
   DFA states to process
10:    for each symbol  $\in \Sigma$  do ▷ For each alphabet symbol
11:      next  $\leftarrow \{\}$ 
12:      accepting  $\leftarrow false$ 
13:      for each state $_\varepsilon \in Q_{DL}[n]$  do ▷ Find the next DFA state
   for a given symbol
14:        for each state  $\in \text{ECLOSE}(E, \text{state}_\varepsilon)$  do
15:          next  $\leftarrow \text{next} \cup \delta_E(\text{state}, \text{symbol})$ 
16:          if  $\delta_E(\text{state}, \text{symbol}) \in F_E$  then
17:            accepting  $\leftarrow true$ 
18:          if next is not empty then ▷ If a next DFA state dis-
   covered
19:            if next  $\notin Q_{DL}$  then ▷ Save the new state, if re-
   quired
20:              add next to  $Q_{DL}$ 
21:              if accepting then ▷ Add the new state to the
   DFA accepting states ar-
   ray if accepting
22:                 $F_D \leftarrow F_D \cup \{\text{next}\}$ 
23:                define  $\delta_D(Q_{DL}[n], \text{symbol}) \leftarrow \text{next}$  ▷ Define transition function
   for a current symbol, cur-
   rent state and the newly-
   discovered next state
24:           $n \leftarrow n + 1$ 
25:    $Q_D \leftarrow \text{set of } Q_{DL} \text{ elements}$ 
26:    $\Sigma \leftarrow \Sigma^+ \setminus \{\varepsilon\}$ 
27:   return  $D$ 

```

Ensure: $D = (Q_D, \Sigma, \delta_D, \iota_D, F_D)$ is a DFA

2.6 DFA Minimization

DEFINITION 9. (Hopcroft et al. 2003) *DFA minimization* is conversion of a given DFA to an equal DFA with a minimal possible number of states.

DFA minimization is a procedure which could further simplify DFA simulation. There are various algorithms to perform the procedure. Since time consumption of DFA minimization is unimportant for the purpose of this thesis, we would further use the algorithm easiest to implement given the algorithms defined in the previous sections: *Brzozowski's Algorithm*.

The Brzozowski's Algorithm is based on the observation made by its author, Janusz Brzozowski (Brzozowski 1963), that reversing a DFA transitions would lead to an NFA for the reversal of the original language whereas further converting it to a DFA using a standard Powerset Construction algorithm produces a minimal DFA for the same reversed language. Moreover, when repeated twice, the same sequence of operations leads to a minimal DFA for the original language.

ALGORITHM 3. DFA reversal

Require: $D = (Q_D, \Sigma, \delta_D, \iota_D, F_D)$ is a DFA

```

1: function REVERSE( $D$ )
2:    $Q_A \leftarrow Q_D$                                 ▷ Copy the old states set to
                                                         the new automaton
3:   for  $state \in Q_D$  do                               ▷ Reverse transitions
4:     for  $symbol \in \Sigma$  do
5:        $\delta_A(\delta_D(state, symbol), symbol) \leftarrow state$ 
6:    $\iota_A \leftarrow F_D$                                 ▷ Let new automaton ini-
                                                         tial states be equal old au-
                                                         tomaton accepting states
7:    $F_A \leftarrow \{\iota_D\}$                             ▷ Let new automaton
                                                         accepting states be equal
                                                         old automaton initial
                                                         state singleton

8:   return  $A$ 

```

Ensure: $A = (Q_A, \Sigma, \delta_A, \iota_A, F_A)$ is an NFA describing the reversal of the language of D

As mentioned above, Brzozowski's Algorithm makes use of two other functions:

DETERMINIZE(E) and REVERSE(D). For their pseudo-code, refer to Algorithm 2 and

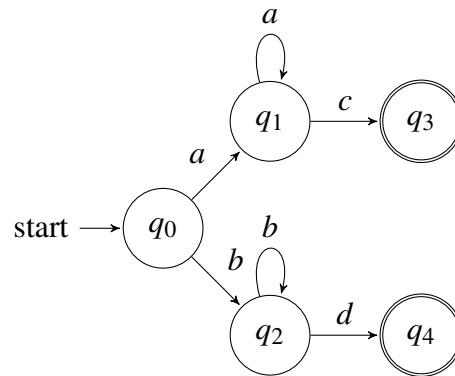


FIGURE 6. A sample non-minimal DFA

Algorithm 3 correspondingly.

ALGORITHM 4. Brzowski's algorithm for DFA minimization

Require: $D = (Q_D, \Sigma, \delta_D, \iota_D, F_D)$ is a DFA

1: **function** MINIMIZE(D)

2: **return** DETERMINIZE(REVERSE(DETERMINIZE(REVERSE(D))))

Ensure: $D_m = (Q_{D_m}, \Sigma, \delta_{D_m}, \iota_{D_m}, F_{D_m})$ is a minimal DFA representing the language of D

EXAMPLE 7. Calling Algorithm 4 with DFA depicted in Figure 6 as argument would return minimal DFA depicted in Figure 10. The algorithm would go through the following steps:

- Receive automaton depicted in Figure 6 as input.
- Reverse automaton depicted in Figure 6 using Algorithm 3 to get automaton depicted in Figure 7.
- Determinize automaton depicted in Figure 7 using Algorithm 2 to get automaton depicted in Figure 8.
- Reverse automaton depicted in Figure 8 using Algorithm 3 to get automaton depicted in Figure 9.
- Determinize automaton depicted in Figure 9 using Algorithm 2 to get automaton depicted in Figure 10.

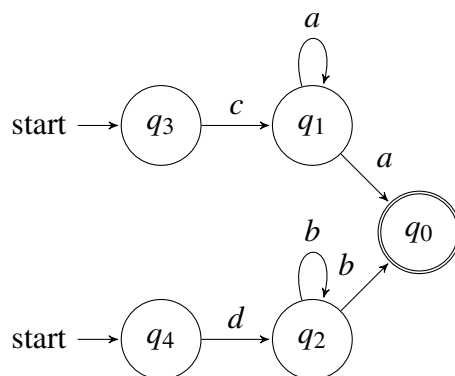


FIGURE 7. An NFA corresponding to the reverse language of DFA depicted in Figure 6

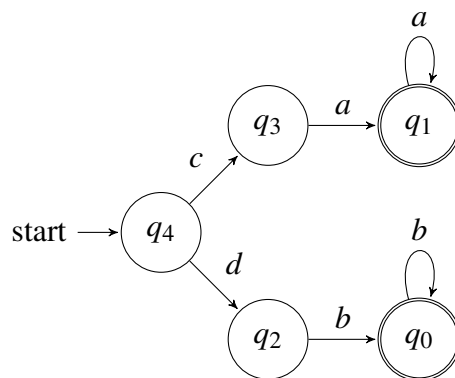


FIGURE 8. A minimal DFA corresponding to the reverse language of the DFA depicted in Figure 6

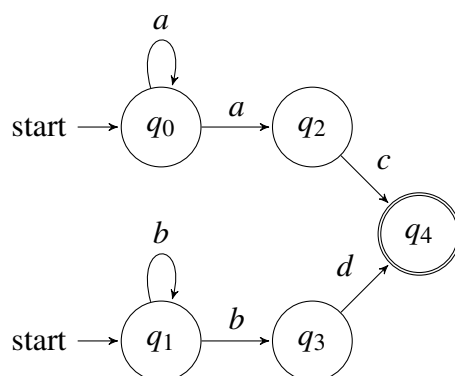


FIGURE 9. A minimal-ending NFA corresponding to the language of DFA depicted in Figure 6

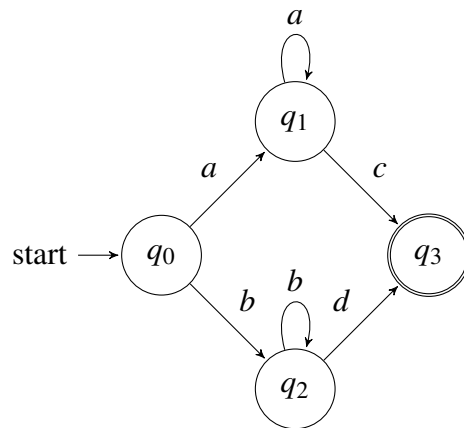


FIGURE 10. A minimal DFA corresponding to the language of the DFA depicted in Figure 6

3 REGULAR EXPRESSIONS

3.1 Overview

Informally, a regular expression is a sequence of characters representing a regular language (recall Definition 4). Regular expressions are said to be equal to finite automata, since the two describe the same regular languages family. The fact of finite automata and regular expressions equality is known as Kleene's theorem (Rozenberg & Salomaa 1997).

THEOREM 2. (Rozenberg & Salomaa 1997) *For every regular expression there is an FA representing the same language.*

Characters of a regular expression over an alphabet encode unions ('+'), concatenations ('·' or no character) and *unbounded repetitions* ('*') of the singleton sets of alphabet characters or other regular expressions over the alphabet (Hopcroft et al. 2003). Parentheses are also used in regular expressions to determine the operations order (Brookshear 1989). We define regular expressions formally:

DEFINITION 10. (Brookshear 1989) *A regular expression over an alphabet Σ is as follows:*

- Empty set \emptyset is a regular expression.
- Empty character ε is a regular expression.
- Every $a \in \Sigma$ is a regular expression.
- If p and q are regular expressions then so is $(p + q)$.
- If p and q are regular expressions then so is $(p \cdot q)$, or (pq)
- If p is a regular expression then so is p^* .

The $*$ -operation is formally given as follows:

DEFINITION 11. (Hopcroft et al. 2003) For a regular expression q , the unbounded repetition, called *Kleene star* q^* , is the infinite union $\bigcup_{i \geq 0} q^i$, where $q^0 = \{\epsilon\}$, $q^1 = q$ and for $i > 1$, $q^i = qq \cdots q$ (a concatenation of i copies of q).

EXAMPLE 8. Below are three sample expanded Kleene-star constructions:

- Let p be a regular expression, then $p^* = \{\epsilon, p, pp, ppp, pppp, \dots\}$.
- Let $q = (a)$, then $q^* = (a)^* = \{\epsilon, a, aa, aaa, aaaa, \dots\}$.
- Let $r = (a + b)$, then $r^* = (a + b)^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$.

EXAMPLE 9. Regular expression $(1(0 + 1)^*)^*10$ describes the set of even binary numbers not dividable by decimal number 4. Using Definition 10, the expression is proven to be regular as follows:

- 0 and 1 are members of Σ , thus, they are regular expressions.
- 0 and 1 are regular expressions, thus so is $(0 + 1)$.
- $(0 + 1)$ is a regular expression, thus so is $(0 + 1)^*$.
- 1 and $(0 + 1)^*$ are regular expressions, thus so is $(1(0 + 1)^*)$.
- $(1(0 + 1)^*)$ is a regular expression, thus so is $(1(0 + 1)^*)^*$.
- $(1(0 + 1)^*)^*$, 0 and 1 are regular expressions, thus so is $(1(0 + 1)^*)^*10$.

Compared to finite automata, regular expressions provide significantly more descriptive and human-readable way of regular language description (Hopcroft et al. 2003). Consequently, they are widely considered a convenient tool for organizing the text search and performing the lexical analysis. One good practical example is *UNIX grep* searching tool, which uses regular-expression-like search pattern notations.

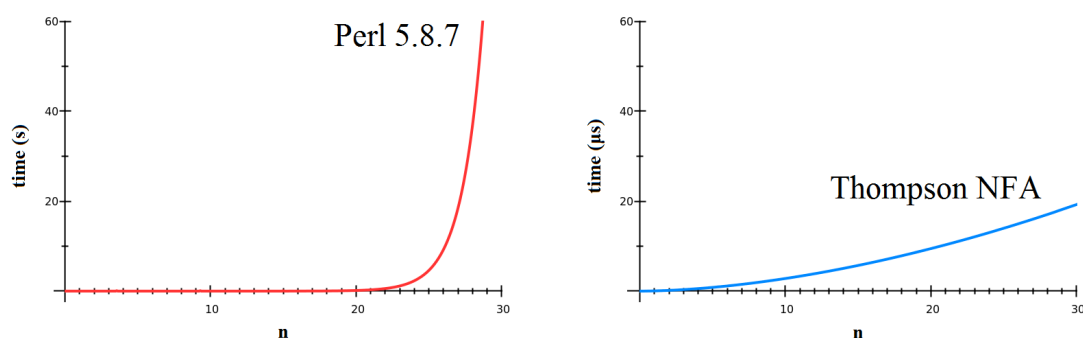


FIGURE 11. Backtracking vs Thompson’s Construction Algorithm executing $(a + \epsilon)^n a^n$ over a^n

3.2 Implementation

There are two major approaches to regular expressions engines implementation: backtracking and Thompson’s construction algorithm. The first one is extremely widely used yet exponentially slow in worst cases. The second one, on the other hand, works in a linear time, but is still seldom used due to historical reasons (Cox 2007).

The canonical example demonstrating the drawback of the backtracking algorithm is a set of regular expressions $(a + \epsilon)^n a^n$, meaning n times $(a + \epsilon)$ followed by n times a , executed over a string a^n (Cox 2007). Figure 11 shows the graphs for the backtracking and Thompson’s construction algorithms execution times for the example depending on the value of n . Due to exponentially rapid growth, the backtracking graph uses seconds as its units, whereas linear Thompson’s graph uses microseconds.

Backtracking

Most of the modern regular expression engines (Python, Perl, Ruby, PHP etc.) use the backtracking approach (Poe 2012). The approach is recursive and based on the simple fall-back strategy. Once a backtracking engine should make a choice, it starts going through all the possible variants, proceeding on success or falling back and picking another variant on fail, until the string is matched (Cox 2007).

While matching a string a^n against the $(a + \epsilon)^n a^n$ pattern, the backtracking engine will

start with a guess that there is an a in place of each $(a + \varepsilon)$ and further go through all the 2^n possible combinations. Example 10 shows such a matching process in detail for $n = 3$.

EXAMPLE 10. (Poe 2012) Matching " a^3 " against $(a + \varepsilon)^3 a^3$ with a backtracking engine would require the following steps:

1. $a_0 a_1 a_2 a a a$
2. $a_0 a_1 a a a$
3. $a_0 a_2 a a a$
4. $a_1 a_2 a a a$
5. $a_0 a a a$
6. $a_1 a a a$
7. $a_2 a a a$
8. $a a a$

Where a_i is an assumption that there is an a in place of $(a + \varepsilon)$ on position number i .

A practical example is Rubular (Rubular 2015), Ruby-based regular expression editor, failing to execute $(a + \varepsilon)^{30} a^{30}$ over a string a^{30} .

Thompson's Construction Algorithm

Thompson's construction algorithm is an algorithm for conversion of regular expressions into equivalent ε -NFA, based on the Kleene's theorem (Thompson 1968).

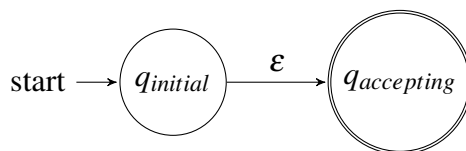


FIGURE 12. A Thompson's ϵ -NFA fragment corresponding to an ϵ -expression

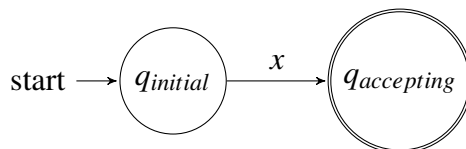


FIGURE 13. A Thompson's ϵ -NFA fragment corresponding to a single-symbol expression; let x be the symbol

Thompson's construction algorithm converts the elements of the initial regular expression to the parts of the new NFA in a recursive way, starting from outer, more complex expressions and recursively proceeding to the inner and simpler ones until discovery of atomic, directly-convertible, single-symbol- or ϵ -expressions. Below is the list of expressions elements and figures depicting their corresponding NFA fragments (Aho et al. 2006):

- Figure 12 shows an ϵ -expression representation.
- Figure 13 shows a single-symbol expression representation.
- Figure 14 shows a union expression representation.
- Figure 15 shows a concatenation expression representation.
- Figure 16 shows a Kleene star expression representation.

The pseudo-code of the algorithm is not presented in this thesis because, due to the nature of the algorithm, the logic of it is easiest to represent as a set of figures combined with a short verbal description, as given above.

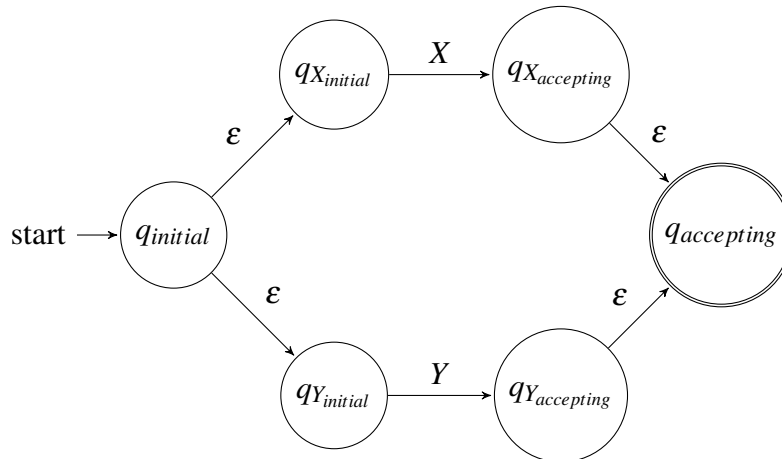


FIGURE 14. A Thompson's ϵ -NFA fragment corresponding to a union expression; let X and Y be automata representing the expressions being united

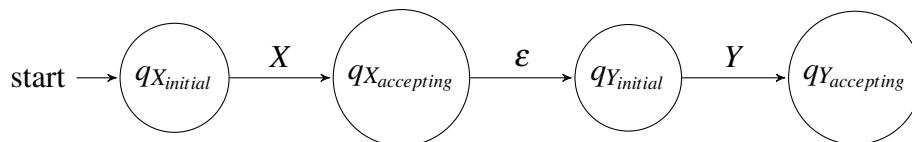


FIGURE 15. A Thompson's ϵ -NFA fragment corresponding to a concatenation expression; let X and Y be automata representing the expressions being concatenated

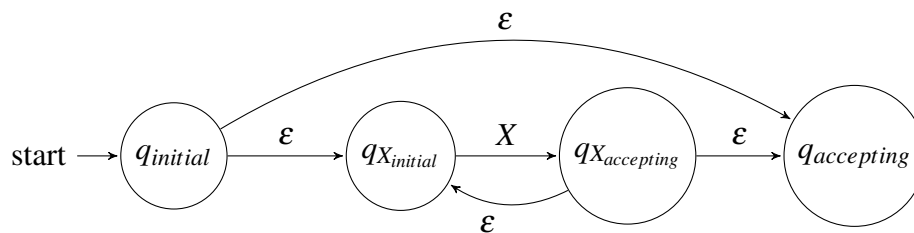


FIGURE 16. A Thompson's ϵ -NFA fragment corresponding to a Kleene star expression; let X be automaton representing the expression under the Kleene star

EXAMPLE 11. Given a regular expression $(a + \epsilon)^3 a^3$ as input, Thompson's construction algorithm will perform the following conversions:

- Each a to the automaton depicted in Figure 17.
- Each ϵ to the automaton depicted in Figure 18.
- $a + \epsilon$ to the automaton depicted in Figure 19.
- $(a + \epsilon)^3$ to the automaton depicted in Figure 20.
- a^3 to the automaton depicted in Figure 21.
- Finally, the initial expression $(a + \epsilon)^3 a^3$ to the automaton depicted in Figure 22.



FIGURE 17. An automaton resulting from conversion of a using the Thompson's construction algorithm

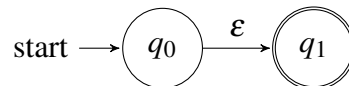


FIGURE 18. An automaton resulting from conversion of ϵ using the Thompson's construction algorithm

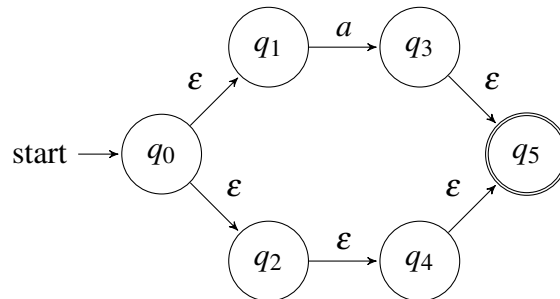


FIGURE 19. An automaton resulting from conversion of ϵ using the Thompson's construction algorithm

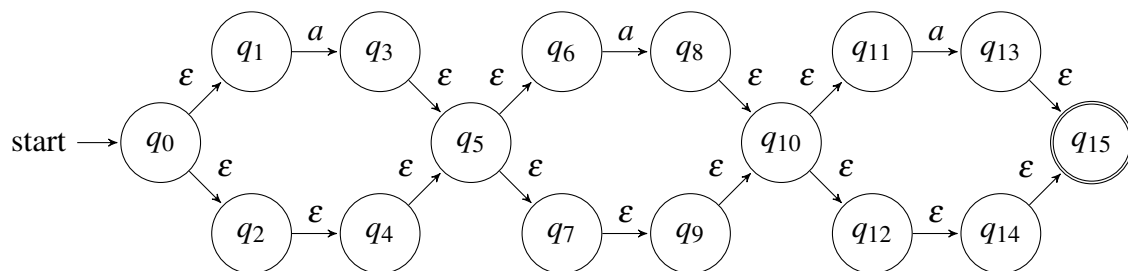


FIGURE 20. An automaton resulting from conversion of $(a + \epsilon)^3$ using the Thompson's construction algorithm

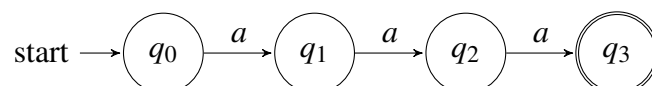


FIGURE 21. An automaton resulting from conversion of a^3 using the Thompson's construction algorithm

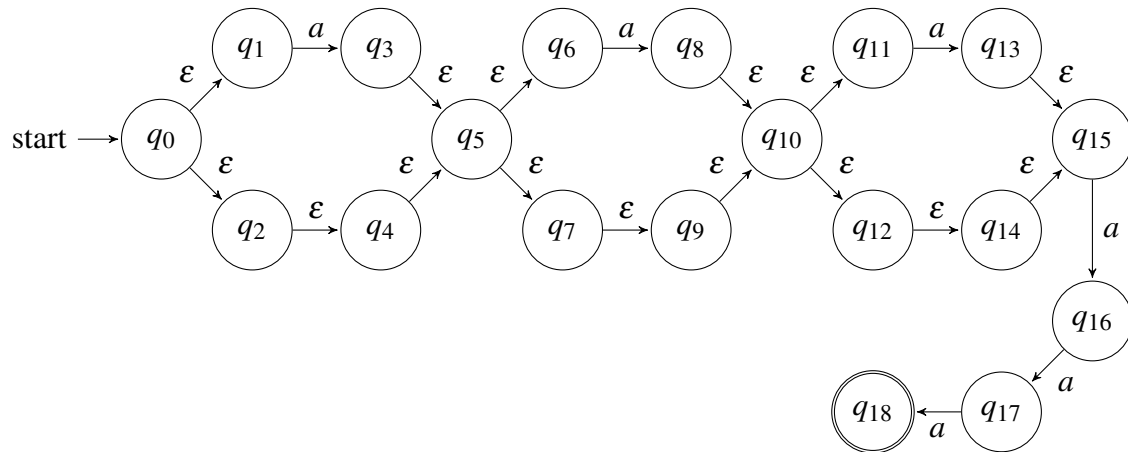


FIGURE 22. An automaton resulting from conversion of $(a + \epsilon)^3 a^3$ using the Thompson's construction algorithm

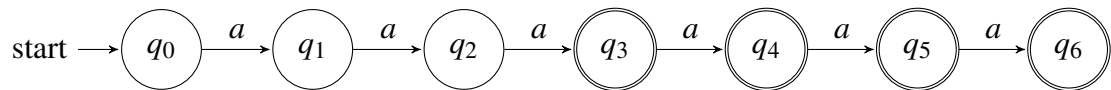


FIGURE 23. An automaton resulting from determinization of the automaton depicted in Figure 22 using Algorithm 2

Since a Thompson's Construction Algorithm output is an ϵ -NFA, it is further possible to convert it to DFA using Algorithm 2. The diagram of a DFA resulting from such a conversion of the above-mentioned ϵ -NFA is shown in Figure 23. Execution of both output ϵ -NFA and further converted DFA over a string a^3 will only take three steps (Cox 2007).

4 COMPLEXITY OF ALGORITHMS

4.1 Overview

Complexity of an algorithm is a measure of the amount of time $T(n)$ and/or space $S(n)$ required for its execution with an input of a given size n (Brookshear 1989). Clearly, input size is not the only parameter affecting the resource requirements of the algorithm; opposite, there are other important parameters such as underlying hardware and compiler characteristics. However, complexity of an algorithm is not entirely synonymous to its exact resource usage. Rather, complexity of an algorithm describes the rates of growth of the resource usage functions of the algorithm, or their *asymptotic behavior* (Arora & Barak 2009). In other words, algorithm complexity analysis only accounts for the fastest-growing term of the polynomial describing the resource usage of the algorithm with respect to an input size, neglecting the constants.

EXAMPLE 12. Time complexity of an algorithm with time requirement of $T(n) = 3n^3 + 125n^2 + 1024n + 3000$ would be n^3 , since it is the fastest growing term in the time requirement function of the algorithm.

Complexity of an algorithm is intended to demonstrate how well it performs in any case (Adamchik 2009). Thus, it is a common practice to calculate the *worst-case complexity* of an algorithm. The worst case is a situation in which resource requirements of the algorithm are maximal for a given input length (Erdelyi 2010).

EXAMPLE 13. Consider Algorithm 5. It checks if some value appears at least twice in a given array. The worst-case scenario for the algorithm occurs when the duplicates are two last elements in the input array. In this case, the algorithm would perform $n - 1$ operation n times, drawing the largest time complexity polynomial term of n^2 . Consequently, it is said that the worst-case time complexity of the algorithm is n^2 .

ALGORITHM 5. Finding duplicate values in an array

Require: A is an array of length n

```

1: hasDuplicate  $\leftarrow false$ 
2: for  $i \in [0, n - 1]$  do
3:   for  $j \in [0, n - 1]$  do
4:     if  $i \neq j$  and  $A[i] = A[j]$  then
5:       hasDuplicate  $\leftarrow true$ 
6:       break
7:   if hasDuplicate then
8:     break
9: return hasDuplicate

```

Ensure: Returns *true* if duplicates present, otherwise *false*

Asymptotic estimation is not perfect: while always being correct in case of sufficiently large inputs, it might be the case that a quadratic algorithm excels a linear one when the inputs are small (Arora & Barak 2009). However, according to Erdelyi in practical settings the input size is typically sufficient, and, therefore, the estimation works well (Erdelyi 2010).

4.2 Algorithm Complexity Metrics

Asymptotic approach serves a basis for several metrics of algorithm complexity, below is the list of the most commonly used of them (Arora & Barak 2009):

- Big-Theta, describing the exact asymptotic behavior of an algorithm
- Big-Oh, describing the upper bound of asymptotic behavior of an algorithm
- Big-Omega, describing the lower bound of asymptotic behavior of an algorithm

Tight complexity of an algorithm

The *tight complexity* of an algorithm, written as $\Theta(f(n))$, where $f(n)$ is a complexity function, and read as *Big-Theta* of $f(n)$, describes the exact asymptotic behavior of the resource usage functions of the algorithm (Arora & Barak 2009). Unlike the further-mentioned Big-O and Big-Omega, Big-Theta is symmetric: if $f(n) = \Theta(g(n))$, then indeed $g(n) = \Theta(f(n))$ (Adamchik 2009).

EXAMPLE 14. Algorithm 5 has two nested loops each with up to n iterations. Therefore, worst case time complexity $T(n)$ of the algorithm is precisely n^2 , not greater nor less than that. Thus, the algorithm is said to have tight worst case complexity of n^2 . Mathematically speaking, $T(n) = \Theta(n^2)$ in the worst case.

Maximal complexity of an algorithm

The *maximal complexity* of an algorithm, written as $O(f(n))$, where $f(n)$ is a complexity function, and read as *Big-Oh*, describes an upper bound of the asymptotic behavior of the resource usage functions of the algorithm (Arora & Barak 2009). Similarly to the further-mentioned Big-Omega, Big-O is not symmetric: $n = O(n^2)$, but $n^2 \neq O(n)$ (Adamchik 2009).

EXAMPLE 15. Worst case time complexity $T(n)$ of Algorithm 5 is $\Theta(n^2)$, which is not greater than $\Theta(n^2)$. Thus, $T(n) = O(n^2)$ and, moreover, $T(n) = O(n^3)$, since $\Theta(n^2)$ is a lower complexity than $\Theta(n^3)$.

Minimal complexity of an algorithm

The *minimal complexity* of an algorithm, written as $\Omega(f(n))$, where $f(n)$ is a complexity function, and read as *Big-Omega*, describes a lower bound of the asymptotic behavior of the resource usage functions of the algorithm (Arora & Barak 2009). Similarly to the Big-Oh, Big-Omega is not symmetric: $n^2 = \Omega(n)$, but $n \neq \Omega(n^2)$ (Adamchik 2009).

EXAMPLE 16. Worst case time complexity $T(n)$ of Algorithm 5 is $\Theta(n^2)$, which is not less than $\Theta(n^2)$. Thus, $T(n) = \Omega(n^2)$ and, moreover, $T(n) = \Omega(n)$, since $\Theta(n^2)$ is a higher complexity than $\Theta(n)$.

Common Complexity Orders

Below is the list of the complexity orders most commonly found in programming practice, from less to more complex, along with their verbal descriptions (Arora & Barak 2009):

1. $O(1)$ — constant
2. $O(\log n)$ — logarithmic
3. $O(n)$ — linear
4. $O(n \log n)$ — "n log n"
5. $O(n^2)$ — quadratic
6. $O(n^3)$ — cubic
7. $n^{O(1)}$ — polynomial
8. $2^{O(n)}$ — exponential

4.3 Algorithm Complexity Analysis

Although the worst-case is the most common type of algorithm complexity analysis, it is not the only one. Below is the complete list of typically used algorithm complexity analysis types (Arora & Barak 2009; Cormen et al. 2001):

- worst-case, describing the maximal possible complexity of an algorithm for any input
- best-case, describing the minimal possible complexity of an algorithm for any input
- average-case, describing the complexity of an algorithm for an average input, which could be hard to determine
- amortized worst-case, describing the guaranteed worst-case complexity of a sequence of operations of any length

EXAMPLE 17. (Cormen et al. 2001) Consider a sequential search algorithm of an array of size n . The algorithm simply picks array items one by one and checks whether or not they match a given element.

- The worst-case time complexity of the algorithm is $O(n)$.
- Its best-case time complexity is $O(1)$.
- Its average-case time complexity is $O(n/2) = O(n)$.

Amortized Worst-Case Complexity

Amortized worst-case complexity is the method of algorithm analysis, dealing with the maximal complexities of consequent executions of an algorithm (Cormen et al. 2001). It is typically used when the complexity of the target algorithm varies as the input size changes. In the amortized analysis, the worst-case complexity of individual executions is estimated and averaged over the amount of executions. Thus, the approach allows to find the close-to-average guaranteed worst-case complexities of execution sequences.

A classical way to demonstrate the usefulness of amortized complexity analysis is a PUSH() method of a generic dynamic array (Tarjan 1985). Whenever a target array has enough free space allocated for an element to push, the algorithm simply adds it to the array, thus working in a constant time. When, on the other hand, the array has not enough space, the algorithm allocates a new, twice larger space for the array, copies an old array to the new space and then adds an element. The total amount of time taken in the latter case is proportional to the array size before PUSH() applied. Consequently, the algorithm worst-case complexity is linear.

As mentioned above, the worst-case time complexity of the PUSH() algorithm is $O(n)$. However, when executed repeatedly, following each worst-case step with a complexity of $O(n)$, the algorithm makes $n - 1$ constant-complexity steps, as shown in Figure 24. At this point, the amortized analysis suggests averaging the complexity over the amount of steps; with $O(n)$ complexity every $n - 1$ steps it would draw an amortized complexity of $O(n/(n - 1)) = O(1)$ (Tarjan 1985). Thus, amortized worst-case time complexity of PUSH() is constant.

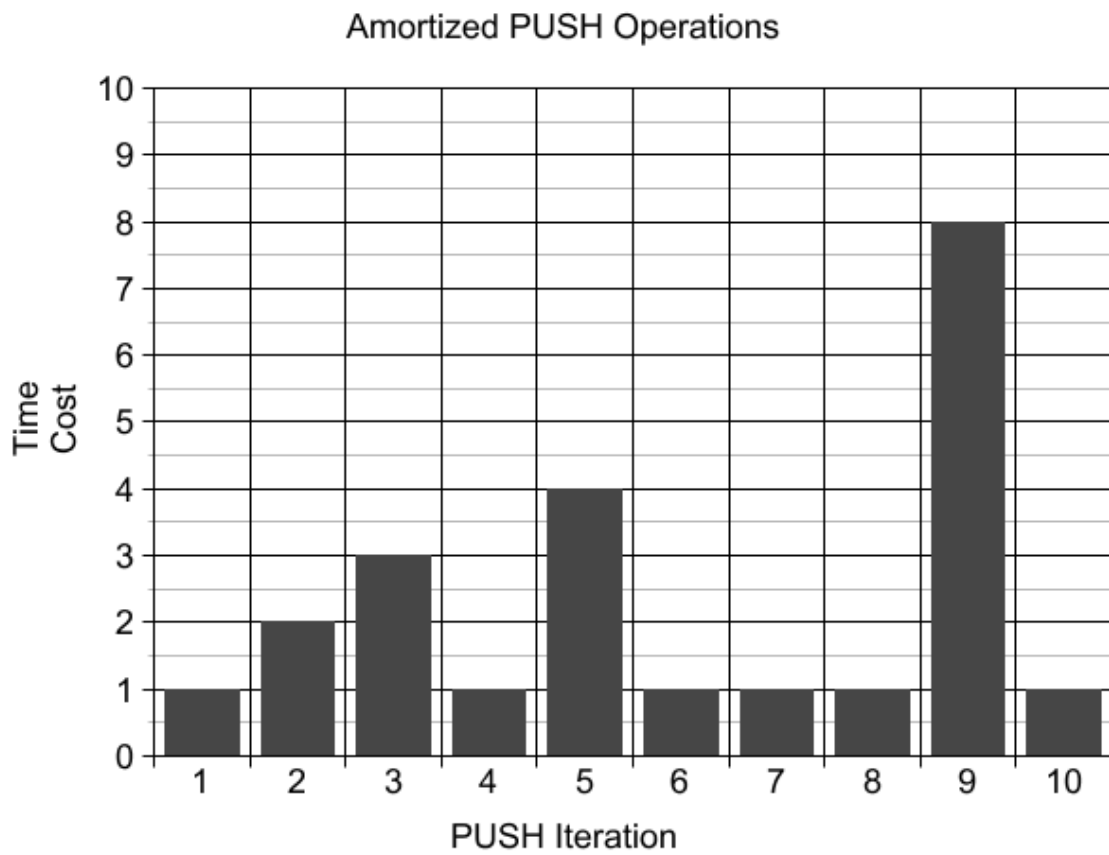


FIGURE 24. Time complexity of dynamic array PUSH() method with respect to an initial array size

5 A GENERIC, JAVASCRIPT-BASED FORMAL LANGUAGE MATCHING FRAMEWORK, WHYNOT.JS

5.1 Overview

Whynot.js, or simply `Whynot`, is an open-source generic real-time formal language matching framework, developed by Stef Busking, lead JavaScript developer of Liones, as one of the core components of Fonto (`Whynot.js` at GitHub 2015). `Whynot` performs the three operations required by Fonto lexical analysis core: validation, completion and extension of a user input.

The framework is conceptually based on the Henry Spencer’s regular expression library: user builds NFA-like programs describing regular languages and further executes them in the framework’s virtual machines against a given input (`Whynot.js` at GitHub 2015; Cox 2007). The functional difference between Henry Spencer’s library and `Whynot` comes from the output. The Henry Spencer’s library only checks validity of inputs. `Whynot`, on the other hand, also suggests complements to invalid inputs turning them into valid ones, if validity is achievable through completion, and useful extensions to valid inputs.

In `Whynot` terms, an *extension* s_e of a string s against a rule R is another string valid against R and consisting of s and some characters inserted to any positions of it. A *useful extension* s_{ue} of a string s against a rule R , where s is valid against R , is a string defined by the following constraints:

1. s_{ue} is an extension of s against R .
2. For n iterations of an unbounded repetition from R in s , there are exactly $n + 1$ additional iterations of the same unbounded repetition in s_{ue} alternated with the original ones, starting from the position before the first original iteration and finishing with the position after the last one.

3. No other string satisfying the constraints 1. and 2. is an extension of s_{ue} against R .

EXAMPLE 18. Running `Whynot` program representing regular expression $(a + (bc))d(e + f)$ with input "d" would return *false* validity flag and traces for completed input representing the following language: $\{ade,adf,bcde,bcdf\}$. The logic behind the output in the case is as follows:

1. "a" and "bc" are the only possible character combinations preceding "d" the rule.
2. "d" is the input, so once discovered it must be used.
3. The input does not entirely match the regular expression, so the validity flag will be set to *false*.
4. "e" and "f" are the two possible characters following "d" in the rule.

With "ad" as input, the same program would return $\{ade,adf\}$, since "bc" does not include "a", and once again a *false* validity flag.

EXAMPLE 19. Running `Whynot` program representing regular expression $(a + b)^*$ with input "a" would return *true* validity flag and traces for extended input representing the following language: $\{aaa,baa,aab,bab\}$. The logic behind the output in the case is as follows:

1. $(a + b)^*$ is an unbounded repetition of a 's and b 's.
2. "a" is an "a" repeated once, so the input is accepted.
3. The input entirely matches the regular expression, so the validity flag will be set to *true*.
4. There is exactly one unbounded repetition iteration present in the input string and exactly one unbounded repetition — in the rule. Thus, all the useful extensions should contain exactly two additional iterations of the same repetition: one before the original and one — after.

With "ab" as input, the same program would return $\{aaaba,aabba,aaabb,aabbb,baaba,babba,baabb,babbb\}$ and a *true* validity flag.

Whynot is a generic framework and only presents a formal language matching level, omitting any application specifics. Thus, complemented with application-specific layers constructing Whynot programs, it has a wide range of possible uses.

5.2 Structure and Operation

Whynot programs consist of instructions of six types:

1. *test*, corresponding to an ϵ -NFA transition.
2. *accept*, corresponding to an accepting state of an ϵ -NFA.
3. *fail*, making the execution to fail in a current thread.
4. *bad*, decreasing the priority of a current thread.
5. *record*, recording a missing symbol in case of a symbol test failure.
6. *jump*, starting a new execution thread for every instruction referenced by the jump.

The programs are executed in multiple threads covering all the possible minimal ways to complete or extend a given input. Upon execution, instructions leave traces storing the threads' history. The traces are grouped into generations, each including all the threads operating on a single input item. The traces left by threads of the last generation are later returned by the framework and allow a user to get all the possible completions or useful extensions of the input. In order to save CPU time, threads arriving to the same instruction within the same generation are merged by the framework. The merge is possible because, being representations of FA, the framework's programs do not depend on the previously visited instructions or previous input.

5.3 Weak Spots and Further Development

The programs of Whynot are executed as built by a user, without any prior optimization. Hence, the amount of test instruction is sub-optimal and, consequently, sometimes branches leading to the different yet non-distinguishable instructions have to be merged. Such merges require significantly more CPU time compared to merges of branches leading to equal instructions. Thus, it is currently assumed that minimization of the instructions amount could be beneficial for the system performance.

There are two possible obvious approaches to the `Whynot` programs optimization and simplification:

1. converting programs to minimal DFA
2. converting programs to minimal ϵ -NFA

According to `Whynot` developer, the first approach is expected to significantly optimize programs in most of the typical use cases due to the straightforward way the programs are constructed, which is close to Thompson's construction algorithm (recall Section 3.2). The second approach, on the other hand, implies the use of sophisticated algorithms, requiring careful examination and lying beyond the scope of bachelor's theses. Moreover, the first approach implementation may later serve as a base for the second one.

For the above-mentioned reasons, this thesis presents an implementation of a new system logically and conceptually based on `Whynot` with the difference between the two being in the underlying computational models. Opposite to generated on-the-fly ϵ -NFA-based programs of `Whynot`, the new system utilizes pre-compiled and shipped to a user in a simple and convenient JSON-based format, minimal DFA.

DEFINITION 12. *JavaScript Object Notation*, or JSON, is a human- and machine- readable data interchange text format based on a subset of the ECMAScript 3 standard. JSON is built on the two structures: a collection of name/value pairs, corresponding to a modern JavaScript object and an ordered list of values corresponding to a modern JavaScript array.

The new system consists of three main subsystems:

1. ϵ -NFA constructor with simple minimal DFA compiler
2. Input completer
3. Input extender

The following chapters provide the detailed overview of the process of the new system development, implementation and testing.

6 COMPILING THE MINIMAL DETERMINISTIC FINITE AUTOMATA

6.1 Overview

This thesis suggested development of a system analogous to `Whynot` yet using simplistic representation of minimal DFA instead of NFA as underlying model for the rules. Thus, the very first step of the system development was creation of a subsystem providing users with means of building the efficiently represented minimal DFA. This chapter describes development of such a subsystem: a DFA compiler.

Since the new system was intended to showcase a possible future development of `Whynot`, it was important to stay as close as possible to `Whynot` architecture. Hence, the DFA compiler was designed to compile the DFA from user-built ϵ -NFA (recall that `Whynot` user-built programs were also representations of ϵ -NFA). The work flow of the DFA compiler was expected to be the following:

1. a user builds an ϵ -NFA representing a required rule
2. the DFA compiler converts the ϵ -NFA into an intermediary minimal DFA
3. the DFA compiler simplifies the intermediary minimal DFA representation to make it as space-efficient as possible and returns it to the user

The stack of technologies used for creation of the DFA compiler as well as the other subsystem of the new system was similar to that of `Whynot`, yet with higher versions of some libraries and standards:

- ECMAScript 5 JavaScript as the main language
- RequireJS 2.1.20 or higher as the files and modules loader
- Mocha 2.3.2 or higher as the main test framework
- Chai 3.2.0 or higher as assertions engine in tests
- Testem 0.9.4 or higher as the test runner

The full code of the DFA compiler along with the other subsystem and unit and perfor-

mance tests could be found at the project's page on Github. (Whynot Premade at GitHub 2015)

6.2 Development and Implementation

To organize the expected work flow, the DFA compiler required the following parts:

- a flexible structure to store intermediary FA before simplification
- a set of methods to build ϵ -NFA
- a simple and serializable format to store the simplified minimal DFA
- a set of methods to perform compilation of a user-built ϵ -NFA into simplistic minimal DFA

Throughout the process of the DFA compiler development, the accent was made on the subsystem flexibility and extensibility rather than performance, since the DFA compiler was expected to be used for creation of pre-compiled minimal DFA and saving them in a JSON-based format out of clients software runtime.

Storing FA Prior to Simplification

The first decision affecting the development of the DFA compiler was about structures to store the intermediary FA during the phases of ϵ -NFA building and ϵ -NFA to minimal DFA conversion. The three available alternatives were:

- a JavaScript object with transitions stored as a transition table
- a JavaScript object with transitions stored as a conventional transition function
- a JavaScript object with transitions stored as a list of individual transitions

The alternatives were differing, in order of appearance, from the most efficient yet least flexible to the least efficient yet most flexible. All the alternatives had four properties in common:

- *initialStates* — an array of numbers of initial states of the FA
- *transitions* — a structure representing transitions of the FA

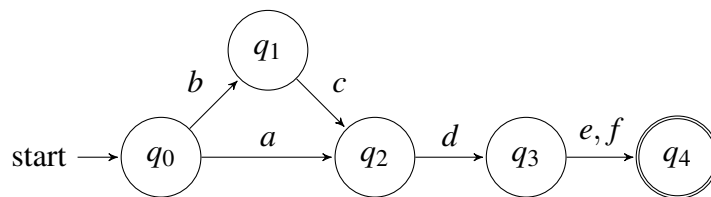


FIGURE 25. A FA corresponding to the regular expression $(a + (bc))d(e + f)$

- *finalStates* — an array of numbers of final states of the FA
- *statesCount* — a total amount of states in the stored FA

The difference between the alternatives was determined by varying *transitions* representations. The first structure had *transitions* represented by an array where each item was describing outgoing transitions of a state with number equal to the index of the item in the array. The items were objects, where each key was corresponding to an alphabet character, and a value contained under the key was an array of numbers of states accessible from the item's state through a transition bound to the character. The structure was the most efficient among all the mentioned in terms of ease of simulation and memory space usage. However, it had two flexibility drawbacks:

1. lack of support for ϵ -transitions support due to forbidden empty-character indices in JavaScript
2. need to modify existing objects rather than add new ones in order to add transitions to the system

The first drawback could have been overcome by adding a non-empty code to describe the empty character. However, it could have required additional overhead and, consequently, could have decreased the convenience of the structure use. The second drawback, in turn, could have been overcome at the cost of increased complexity of the methods relying on the structure. Thus, structures with higher flexibilities were preferred over this.

EXAMPLE 20. The Finite automaton from Figure 25 represented as a JavaScript object with transitions stored as a transition table.

```

{
  "initialStates": [0],

```

```

"transitions": [
  {"a": [2], "b": [1]},
  {"c": [2]},
  {"d": [3]},
  {"e": [4], "f": [4]}
],
"finalStates": [4]
"statesCount": 4
}

```

The second structure had *transitions* represented by an array, where each element contained information of a transition or a set of transitions allowing to get from a state to another. Each transition element had the three properties:

1. *stateFrom* — a state from which the transition(s) originate(s)
2. *stateTo* — a state to which the transition(s) lead(s)
3. *characters* — an array of characters bound to transitions connecting *stateFrom* to *stateTo*, one character per transition

The structure was free of the first drawback of the previous structure. However, the second drawback still applicable. Similarly to the preceding case, more flexible structures were preferred.

EXAMPLE 21. The Finite automaton from Figure 25 represented as a JavaScript object with transitions stored as a conventional transition function.

```

{
  "initialStates": [0],
  "transitions": [
    {"stateFrom": 0, "stateTo": 1, "characters": ["b"]},
    {"stateFrom": 0, "stateTo": 2, "characters": ["a"]},
    {"stateFrom": 1, "stateTo": 2, "characters": ["c"]},
    {"stateFrom": 2, "stateTo": 3, "characters": ["d"]}
  ]
}

```

```

    {"stateFrom": 3, "stateTo": 4, "characters": ["e","f"]}
  ],
  "finalStates": [4],
  "statesCount": 4
}

```

The third structure had *transitions* represented by an array, where each element contained information of a single transition allowing to get from a state to another. The properties of the elements were as follows:

1. *stateFrom* — a state from which the transition originate
2. *stateTo* — a state to which the transition lead
3. *character* — a character bound to a transition connecting *stateFrom* to *stateTo*

The structure was highly-flexible as it used exactly one object per transition. It was the most convenient for manipulation among all the presented options; thus, it was chosen. The structure will be referred to as an intermediary FA further in this thesis.

EXAMPLE 22. The Finite automaton from Figure 25 represented as a JavaScript object with transitions stored as a list of individual transitions.

```

{
  "initialStates": [0],
  "transitions": [
    {"stateFrom": 0, "stateTo": 1, "character": "b"},
    {"stateFrom": 0, "stateTo": 2, "character": "a"},
    {"stateFrom": 1, "stateTo": 2, "character": "c"},
    {"stateFrom": 2, "stateTo": 3, "character": "d"},
    {"stateFrom": 3, "stateTo": 4, "character": "e"},
    {"stateFrom": 3, "stateTo": 4, "character": "f"}
  ],
  "finalStates": [4],
  "statesCount": 4
}

```

}

Building Intermediary FA

Thompson's construction algorithm (recall Section 3.2) represented a minimal well-defined framework for creation of FA for any regular expression. The methods it included were as follows:

- create a single-character (single-transition) FA
- create a concatenation of two FA
- create a union of two or more FA
- create a unbounded repetition (Kleene-star) of a FA

Since, according to Theorem 2, every FA could be expressed as a regular expression, the four methods allowed for creation of any possible FA. Thus, FA building tools comprised these methods.

Converting Intermediary FA to Minimal DFA

The next step after building FA was their conversion to minimal DFA, that is their determinization and minimization. Since convenience was the main factor affecting the development choices during the NFA compiler development, the most straightforward ways of FA determinization and minimization were chosen: Schneider's algorithm (Algorithm 2) and Brzowski's algorithm (Algorithm 4) correspondingly. Particularly, the second one had an analog with significantly lower computational complexity, Hopcroft's algorithm (Hopcroft et al. 2003). However, Brzowski's algorithm made use of Schneider's algorithm which was the most convenient option to solve the case, whereas Hopcroft's algorithm implementation required an extra effort. Thus, Brzowski's algorithm was chosen.

Simplifying DFA Representation

Opposite to intermediary FA case, the choice of a structure to store the resulting DFA was driven by factors of space-efficiency and ease of simulation. The comparison of structures to store intermediate FA presented earlier in this chapter suggested that the most space-efficient and easiest to simulate structure was the FA with transitions stored as a transition table. However, the comparison was dealing with FA in general, including NFA. Since at this point of the system development process the decision had to be made about DFA storage, the structure was further lightened to store only the information required by DFA. The resulting structure properties were as follows:

1. *initialState* — a number of the initial state of the stored FA
2. *transitions* — a structure to store transitions
3. *finalStates* — a set of final states of the stored FA
4. *statesCount* — a total amount of states in the stored FA

The lightened structure had *transitions* representation close to that of its prototype as a transition with the difference in the values contained under character keys: instead of arrays of numbers of states, the values were numbers of states themselves, since for DFA it was not possible to have more than one transition with a particular character going from a state to another.

EXAMPLE 23. The DFA from Figure 25 represented as a JavaScript object with transitions stored as a transition table.

```
{
  "initialState": 0,
  "transitions": [
    {"a": 2, "b": 1},
    {"c": 2},
    {"d": 3},
    {"e": 4, "f": 4}
  ],
}
```

```
"finalStates": [4]
"statesCount": 4
}
```

6.3 Suggested Further Development

The further development of DFA compiler will have least two possible directions:

1. changing output from minimal DFA to minimal NFA
2. adding the output FA meta-data allowing for easier simulation by the system

The first direction will require changing the NFA determinization and DFA minimization step to sole NFA minimization. The second one will require methods to analyze the output FA and a change to the output structure allowing for the storage of the newly acquired meta-data. Both of the methods will have a potential of decreasing the output FA simulation complexity, which, however will have to be examined in a separate research or researches.

7 COMPLETING AND EXTENDING THE INPUT

7.1 Overview

The second subsystem of the system suggested by this thesis was intended to find completions or extensions of a given string against a given DFA. The subsystem was called "DFA Traverser" after the core component required for both completing and extending the input.

The work flow of the subsystem was expected to be as follows:

1. A user inputs a string and a DFA.
2. The traverser component starts traversing the DFA beginning with its initial state.
3. For every new transition it traverses, it performs a check depending on the mode.
 - a) In completion mode, it checks whether or not the new transition added to the preceding transitions minimally completes the input.
 - b) In extension mode, it checks whether or not the new transition usefully extends the input.
4. The system returns an array of input completions or extensions correspondingly.

7.2 Development and Implementation

To make the system comply with the expected work flow, the subsystem development consisted of building the three components:

1. the core component, DFA traverser
2. the input-completing language filter component
3. the input-extending language filter component

The class built to represent the DFA traverser along with completion and extension filters had of the following core parameters:

- `initialState` — initial state of the rule-defining DFA
- `transitions` — transition table of the DFA
- `transposedTransitions` — transposed (in a matrix fashion) transition table of the DFA
- `finalStates` — final states of the DFA
- `finalRecord` — array of records considered final during the DFA execution

Furthermore, the class was served by several core methods:

- `transposeTransitions` — converted transition to transposed transitions.
- `findInsertionIndex` — similarly to the original `Whynot`, returned index for a record to be inserted to a level's final records array in order to keep the array sorted by the amount of missing records
- `insertNewTailRecord` — inserted a new level-final record
- `insertNewRecord` — inserted a new non-level-final record
- `processTailRecords` — processed a single level
- `execute` — traversed the DFA to complete or extend the input

The new terms used in the above list, such as (missing/accepted) record, level, level-final, non-level-final are explained in the further subsections.

Storing the DFA execution traces

The DFA traverser required a data type to store traversal history during and after the execution. To achieve a higher level of flexibility, the decision was made to store it in unidirectional linked lists, where items were describing transitions along with meta-information. Thus, `Record` class was created (instances referred to simply as records further in the chapter) to represent an element of such a list. The properties of the type were decided to be as follows:

- `accepted` — an indicator showing whether the transition(s) described by the record are accepted or not
- `previousRecord` — a link to the previous item in the list
- `targetState` — a state to which the described transition(s) led
- `characters` — an array of characters bound to the described transition(s); always a single character for an accepted transition, but could be multiple for a missing one
- `missingCount` — amount of records with missing transitions in the list up to this record inclusive
- `acceptedCount` — amount of records with accepted transitions in the list up to this record inclusive

The data type represented a merger of `Thread` and `Trace` data types of the original `Whynot`. Besides the above-mentioned properties, `Record` had several methods suited for operations on both individual records and lists of them. The most important methods were as follows:

- `isExtensionOf` — checked whether or not the caller record was a head of a list uselessly extending (non-minimally) the other, head-passed-by-argument list.
- `isPartialOf` — checked whether or not the set of characters of the caller record is a subset of characters of an argument record.
- `hasLoops` — checked whether or not there was a character repeat with constant `acceptCount` in the list headed by the caller record.

The properties and methods mentioned above were developed to support the input completion feature of the subsystem, but not the input extension. However, effort required to implement the input extension was limited slight modifications to the `isExtensionOf` and `hasLoops` methods.

Minimal DFA Preprocessing

In order to save the multiple-transition records in a rapid and convenient manner, a pre-processed copy of transition tables was required. The preprocessing was done using the

algorithm which was called "DFA transitions transpose" after the similarity of the operation over DFA and matrix transpose.

ALGORITHM 6. DFA transitions transpose

Require: transitions is a transition table of a minimal DFA

```

1: function TRANSPOSETRANSITIONS(transitions)
2:   transpTransitions  $\leftarrow$  []
3:   for stateNumber  $\in$  [0, transitions.length) do
4:     tranpTransitions[stateNumber]  $\leftarrow$  new Map()
5:     for (character, nextState)  $\in$  transitions[stateNumber] do
6:       if transpTransitions[stateNumber][nextState] is not defined then
7:         transpTransitions[stateNumber][nextState]  $\leftarrow$  []
8:         PUSH(transpTransitions[stateNumber][nextState], character)

```

Verbally, the algorithm operation could have been expressed as follows: it was creating an additional transition table answering not the question "where does a particular character lead from a given transition?", but "which characters do lead to a particular transition from a given one?".

The resulting transposed transition were capable of decreasing the amount of branches created by the system. For instance, consider an automaton A , which has the following parameters:

1. $Q = \{q_0, q_1, q_2\}$
2. $\Sigma = \{a, b, c, d, e, f\}$
3. δ is defined in the following way: $\delta(q_0, a) = q_1$, $\delta(q_0, b) = q_1$, $\delta(q_0, c) = q_1$, $\delta(q_0, d) = q_1$, $\delta(q_0, e) = q_1$, $\delta(q_1, f) = q_2$
4. $\iota = q_0$
5. $F = \{q_1\}$

When traversed using the conventional transitions against some input f , the automaton will create five separate branches, one for each of transitions from q_0 to q_1 . In case of transposed transitions usage, however, the system will only require a single transition including a transposed branch. The characters and the accepted flag will be `characters = [a, b, c, d, e]` and `accepted = false` for the first record and `characters = [f]` and `accepted = true` for the second one correspondingly.

Traversing the DFA

The main decision at this step of development was regarding approach to the order of traversal. The two options were:

1.
 - a) Go through DFA transitions one-by-one, creating equal-length branches, regardless of whether a current transition is bound to the currently processed character from the input string or not.
 - b) If a branch arrives to a final state having all the input processed, save its tailing (last) record as a final record, removing it from further execution.
 - c) Return final records to the user.
2.
 - a) Align the traversal by input characters through breaking it up the into levels, each corresponding to a character, and within each level execute branches in the fashion similar to that of item 1.
 - b) Save the records for transitions bound to the current input character as final records for the level.
 - c) Include the final records of the level to the records list of the next level in order to continue execution in the next level with them.
 - d) Once the input is over, go through another level having input character set to *null* and save all the records arriving to the final state as final records, removing them from further execution.
 - e) Return final records to the user.

Among the two options, the first one did not provide the extremely convenient feature of character-aligned traversal. On the other hand, the second one did not just provide the character alignment, but was also extremely close to the original *WhyNot*. For these two reasons, the second approach was chosen.

Completing the Input

The input completion component was represented by a filtering method checking for every newly-added non-level-final record if it is free from loops and not an extension of a

characters chain headed by some other record existing in the level. Both loop and extension checks have been implemented as Record class methods. The component was capable of returning only those heads of records chain which had their chains including all the input characters in order and were finishing in final states.

EXAMPLE 24. Executing DFA representing the regular expression $(a + (bc) + (pbcx)) \cdot d(e + f)$ with input "d" returned the following language: $\{ade, adf, bcde, bcdf\}$. The logic behind the output in the case was as follows:

1. "a", "bc" and "pbcx" are the three possible character combinations preceding *d* the rule.
2. "pbcx" is an extension of "bc"; thus, it should not be returned.
3. "d" is the input, so once discovered it must be used.
4. "e" and "f" are the two possible characters following "d" in the rule.

During the completion component development, the detailed observation of the original `Whynot` has been done. It has been found that `Whynot` does not always return the minimal completions of the input. Particularly, the test from the Example 24 returned the following results $\{ade, adf, bcde, bcdf, pbcxde, pbcxdf\}$, where "pbcxde" and "pbcxdf" were extensions of "bcde" and "bcdf" correspondingly. The fix of this behavior was important in terms of the framework conforming to its declared operation, but not critical for the Liones case; hence, it was not further considered within the scope of this thesis.

Merging the Branches

In the original `Whynot` the execution branches were merged not to execute the same instruction several times against the same input item, but the merges were presumably inefficient due to the suboptimal amounts of states in the `Whynot` programs. It was assumed that reduction of the amount of states in the rules may have increased the merges performance, and, consequently, speeded up the system. The merging for the new system was developed as following: if for a new record there was another record arriving to the same state earlier in the level, but the list headed by the newer record's does not extend that of the older record, the records were merged.

There were two approaches to storage of merged records:

1. An array of merged-in records in each record.
2. Multiple previous records instead of single ones in each record.

Similarly to the traversal case, the second choice was closely reassembling the approach of original *Whynot*; thus, it was preferred to the first one.

Extending the Input

The input extension system required slightly more than a minor changes to the loop- and extension- checking methods: the first method had to be changed to force single missing-records loop iterations to be inserted wherever possible, and the second method — to check for the longest possible extension instead of the shortest one. The three other limited changes required to extend the input were:

1. Add to *Record* a method to check for loops with accepting records.
2. Use the method in item 1. to check for such loops during the automaton traversal, and, in case of such a loop discovery, go back to the record beginning the loop, insert there a fully-missing loop and further recalculate the missing counter for the records list part corresponding to the loop with accepted records.
3. Add to the DFA traverser an input flag determining whether or not an input string is accepted by an input rule.

For the user, the operation of the system was expected to be nearly preserved after the changes were done. The only deviation from the completion-only system work flow was in the use of the flag determining whether or not the input was valid against an input rule.

The input extension component has been developed but not implemented within during the research due to the time restrictions.

7.3 Systems Comparison

Due to the improperly chosen structure for storing the intermediary results, implementation of the merging of branches for the tests with rules including unbounded repetition was impossible at this stage. Thus, the new system complexity was higher, and comparing systems performances using tests with unbounded repetition was not feasible. From the functional point of view, however, the new system without merges was flawlessly completing the input not replicating the earlier-described (see Subsection 7.2) non-minimal behavior of the original `Whynot`.

The comparison of the original `Whynot` and the traverser and completion component of new system was done using a series of three simple tests. Each test consisted of a rule without unbounded repetitions and an input string, and was run for exactly 1000 iterations, in order to amortize (recall Subsection 4.3) the working time of JavaScript internal mechanisms responsible for memory management. Below is the list of tests in order of increasing complexity. Each test goes along with the times of the frameworks to run 1000 iterations of it:

1. $(a + (bc))d(e + f)$ against "d":
 - original `Whynot` runtime: 97ms
 - new system runtime: 25ms
 - the new system is 3.9 times faster in the case
2. $(a + (bc) + (pbcx))d(e + f)$ against "d":
 - original `Whynot` runtime: 140ms
 - new system runtime: 30ms
 - the new system is 4.6 times faster in the case
3. $(a + (bc))d(e + f)g(m + k)$ against "dg":
 - original `Whynot` runtime: 170ms
 - new system runtime: 30ms
 - the new system is 5.6 times faster in the case

It was clear from the results that with the increase of the test complexity and the amount of

merges required by the test, the runtime of the original `Whynot` began to grow faster than that of the new system. Hereby, it was concluded that the complexity of original `Whynot` was higher in tests without unbounded repetition. Furthermore, the new system was also theoretically expected to have lower complexity in cases with unbounded repetition under a certain assumption. The assumption was that the only difference between the cases without unbounded repetition and those with it was in the larger amount of merges in the latter.

7.4 Further Development

The further development of the subsystem required the deep analysis of the Automata Theory, including the exploration of automata types other than FA. This task lacked a larger amount of research time and resources; thus, it was outside the scope of Bachelor's theses.

8 CONCLUSIONS

The theoretical purpose of the thesis was to explore the concepts of Finite Automata, Regular Languages and Algorithm Complexity. The practical purpose, on the other hand, was to build a DFA-based version of `Whynot`, a generic JavaScript-based language matching framework. During the research, the new system has been built, but it was not, unfortunately, entirely completed. However, the system completion degree was enough to handle an efficient research, and two positive results were obtained.

The first positive result was that the answer to the main practical question has been found using the cases not including the unbounded repetition. The comparative testing of the systems showed that not just the constant times were better in the new system, but the complexity was lower too. Moreover, the cases with the unbounded repetition were theoretically expected to benefit more from the underlying structure change than those without it. The expectation of the benefit for the cases with unbounded repetitions was based on a particular assumption. The assumption was that the only consequence of the addition of the unbounded repetitions was the increased amount of merges. Thus, it was concluded that changing the original `Whynot` underlying structure from NFA to DFA was not just possible but feasible, as it led to the system complexity decrease and, consequently, performance profits. Furthermore, introduction of the transposed transitions was additionally decreasing the system complexity. Still, an important fact was that the system complexity was remaining exponential, but the polynomial in the power of the complexity exponent was decreasing in rates of growth.

The second positive result was that the original `Whynot` completion system was not always handling the results in a minimal fashion, so that non-minimally extended and duplicate results were left in particular test cases. According to `Whynot` developer, however, changing the system to operate in a minimal fashion was not necessary for the needs of `Liones` and, moreover, could have brought not just additional accuracy but also extra complexity to the framework.

BIBLIOGRAPHY

Adamchik, Victor S. 2009. Algorithmic Complexity. WWW-document. <http://www.cs.cmu.edu/~adamchik/15-121/lectures/Algorithmic%20Complexity/complexity.html>. Updated 18.09.2010. Referred 13.07.2015.

Aho, Alfred V., Lam, Monica S., Sethi, Ravi & Ullman, Jeffrey D. 2006. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley, Boston, MA, USA.

Arora, Sanjeev & Barak, Boaz 2009. Computational Complexity: A Modern Approach. Cambridge University Press, Cambridge, England.

Brookshear, Glenn J. 1989. Theory of Computation: Formal Languages, Automata, and Complexity. The Benjamin/Cummings Publishing Company, Inc, Redwood City, CA, USA.

Brzozowski, Janusz A. 1963. Canonical regular expressions and minimal state graphs for definite events. Proc. Sympos. Math. Theory of Automata (New York, 1962), pages 529–561.

Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L. & Stein, Clifford 2001. Introduction to Algorithms, Second Edition. The MIT Press, Cambridge, MA, USA.

Cox, Russ 2007. Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...). WWW-document. <https://swtch.com/~rsc/regexp/regexp1.html>. Updated 02.02.2015. Referred 02.07.2015.

Erdelyi, Arthur 2010. Asymptotic Expansions (Dover Books on Mathematics). Dover Publications, Mineola, NY, USA.

FontoXML 2015. Website. <http://www.fontoxml.com/>. No Update Information. Referred 21.04.2015.

Hopcroft, John E., Motwani, Rajeev & Ullman, Jeffrey D. 2003. Introduction to Automata Theory, Languages, and Computation, Second Edition. Pearson Education International Inc., Upper Saddle River, NJ, USA.

Liones 2015. Website. <http://www.liones.nl/>. No Update Information. Referred 01.07.2015.

Poe, Curtis 2012. Beginning Perl. Wiley / Wrox, Ebook.

Rabin, Michael O. & Scott, Dana S. 1959. Finite automata and their decision problems. IBM Journal of Research and Development, 3(2):114–125.

Reghizzi, Stefano C. 2009. Formal Languages and Compilation (Texts in Computer Science). Springer, London, United Kingdom.

Rozenberg, Grzegorz & Salomaa, Arto 1997. Handbook of Formal Languages: Volume 1. Word, Language, Grammar. Springer, New York, NY, USA.

Rubular 2015. Website. <http://rubular.com/>. No Update Information. Referred 02.10.2015.

Schneider, Klaus 2003. Verification of Reactive Systems: Formal Methods and Algorithms (Texts in Theoretical Computer Science. An EATCS Series). Springer, New York City, NY, USA.

Silmaril Consultants 2015. Online Documentation. <http://xml.silmaril.ie/>. Updated 2015. Referred 30.10.2015.

Tarjan, Robert E. 1985. Amortized computational complexity. SIAM Journal on Algebraic Discrete Methods, 6(2):306–318.

Thompson, Ken 1968. Programming techniques: Regular expression search algorithm. Commun. ACM, 11(6).

Whynot Premade at GitHub 2015. Git Repository. <https://github.com/dnl-blkv/whynot-premade>. Updated 05.11.2015. Referred 06.11.2015.

Whynot.js at GitHub 2015. Git Repository. <https://github.com/bwrrp/whynot.js>. Updated 21.08.2015. Referred 29.10.2015.

World Wide Web Consortium 2015. Extensible Markup Language (XML). WWW-document. <http://www.w3.org/XML/>. Updated 19.05.2015. Referred 29.10.2015.