Preface

This report forms the Master’s Thesis of A.A. de Beijer from the Computing Science department of the Eindhoven University of Technology in The Netherlands. The thesis is a research monologue in the field of automata theory. In this thesis we present two new transformation operations on finite state automata, called stretching and jamming. These transformations are intended to increase the performance of the automata.

Readers who are mainly interested in the theoretical side of the transformations are referred to chapters 2 and 3. An overview of the abstract algorithms that model the transformations is given in chapters 4 and 5. Implementation details can be found in chapter 6. If the reader is interested in the practical results, and wants to know in which cases stretching and jamming are useful, we refer to chapter 7.

I had the pleasure to work on this thesis at the University of Pretoria in South Africa. I would like to thank Prof. Dr. Derrick Kourie from the University of Pretoria for his supervision and inspiration during my stay in South Africa. I also would like to thank Prof. Dr. Bruce Watson for his supervision and inspiration. Lastly, I want to thank Dr. Ir. Alex Telea for his advice and Ir. Loek Cleophas who helped me with several revisions of this thesis.
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td>Summary</td>
<td></td>
<td>vii</td>
</tr>
<tr>
<td>0</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Preliminaries</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Definitions of Stretching and Jamming</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Bit-level Stretching and Jamming</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>Algorithms</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>Improvements on the Algorithms</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>Implementation</td>
<td>31</td>
</tr>
</tbody>
</table>

### 2 Definitions of Stretching and Jamming

- 2.0 General Definitions
- 2.1 Stretching and Jamming by a Factor $f$

### 3 Bit-level Stretching and Jamming

- 3.0 Theoretical Results for Stretching
- 3.1 Theoretical Results for Jamming

### 4 Algorithms

- 4.0 Stretch Algorithm
- 4.1 Jam Algorithms

### 5 Improvements on the Algorithms

- 5.0 New Stretch Algorithms
- 5.1 New Jam Algorithms

### 6 Implementation

- 6.0 Overview
- 6.1 Classes
  - 6.1.0 Vec
  - 6.1.1 Matrix
  - 6.1.2 TransitionTable
  - 6.1.3 FA
  - 6.1.4 DFA
  - 6.1.5 NFA
  - 6.1.6 JammedDFA
# 7 Benchmarking

## 7.0 Overview

## 7.1 Generation of Random DFAs

## 7.2 Generation of Random Paths

## 7.3 Performance Results for Stretching

## 7.4 Performance Results for Jamming

## 7.5 Conclusions

# 8 Conclusions and Future Work

# A Source code
Table of Figures

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>Stretching transition ((p, a, q)) into (k) sequential transitions.</td>
</tr>
<tr>
<td>2.1</td>
<td>DFA (FA_0)</td>
</tr>
<tr>
<td>2.2</td>
<td>NFA (FA_1), a 2-stretch of DFA (FA_0)</td>
</tr>
<tr>
<td>2.3</td>
<td>Injection (\tau)</td>
</tr>
<tr>
<td>2.4</td>
<td>Bijection (\varphi)</td>
</tr>
<tr>
<td>3.0</td>
<td>DFA (FA_0)</td>
</tr>
<tr>
<td>3.1</td>
<td>Transition table of DFA (FA_0)</td>
</tr>
<tr>
<td>3.2</td>
<td>NFA (FA_1), a '2-stretch' of DFA (FA_0)</td>
</tr>
<tr>
<td>3.3</td>
<td>Transition table of NFA (FA_1)</td>
</tr>
<tr>
<td>3.4</td>
<td>DFA (FA_2), determinized NFA (FA_1)</td>
</tr>
<tr>
<td>3.5</td>
<td>Transition table of DFA (FA_2)</td>
</tr>
<tr>
<td>3.6</td>
<td>Break-even graphs for stretching</td>
</tr>
<tr>
<td>3.7</td>
<td>Break-even graphs for jamming</td>
</tr>
<tr>
<td>3.8</td>
<td>Stretching and Jamming</td>
</tr>
<tr>
<td>5.0</td>
<td>Minimization after a (d)-stretching operation</td>
</tr>
<tr>
<td>5.1</td>
<td>NFA (FA_0 = ({a, b, c}, \mathbb{B}, \delta_0, a, {c}))</td>
</tr>
<tr>
<td>5.2</td>
<td>NFA (FA_{1a}) and NFA (FA_{1b})</td>
</tr>
<tr>
<td>5.3</td>
<td>DFA (FA_2 = ({a, b}, \mathbb{B}^2 \cup {0}, \delta_2, a, {c}))</td>
</tr>
<tr>
<td>5.4</td>
<td>NFA (FA_0) and NFA (FA_1), a 2-jam of (FA_0)</td>
</tr>
<tr>
<td>6.0</td>
<td>Class diagram</td>
</tr>
<tr>
<td>7.0</td>
<td>Memory usage for random 2-bit DFAs of 100 states</td>
</tr>
<tr>
<td>7.1</td>
<td>Memory usage for random 2-bit DFAs of 1000 states</td>
</tr>
<tr>
<td>7.2</td>
<td>Memory usage for random 4-bit DFAs of 10 states</td>
</tr>
<tr>
<td>7.3</td>
<td>Memory usage for random 4-bit DFAs of 100 states</td>
</tr>
<tr>
<td>7.4</td>
<td>Memory usage for random 4-bit DFAs of 1000 states</td>
</tr>
<tr>
<td>7.5</td>
<td>Memory usage for random 8-bit DFAs of 10 states</td>
</tr>
<tr>
<td>7.6</td>
<td>Memory usage for random 8-bit DFAs of 100 states</td>
</tr>
<tr>
<td>7.7</td>
<td>Memory usage for random 8-bit DFAs of 1000 states</td>
</tr>
<tr>
<td>7.8</td>
<td>Memory usage for random 8-bit DFAs of 1000 states, MSB first</td>
</tr>
<tr>
<td>7.9</td>
<td>String processing time for random 8-bit DFAs of 100 states, 10% density</td>
</tr>
<tr>
<td>7.10</td>
<td>String processing time for random 8-bit DFAs of 100 states, 50% density</td>
</tr>
<tr>
<td>7.11</td>
<td>String processing time for random 8-bit DFAs of 1000 states, 10% density</td>
</tr>
<tr>
<td>7.12</td>
<td>String processing time for random 4-bit DFAs of 100 states, 10% density</td>
</tr>
<tr>
<td>7.13</td>
<td>Memory usage for random 1-bit DFAs of 100 states</td>
</tr>
</tbody>
</table>
7.14 Memory usage for random 1-bit DFAs of 1000 states .................. 62
7.15 Memory usage for random 2-bit DFAs of 100 states .................. 63
7.16 Memory usage for random 2-bit DFAs of 1000 states ................. 63
7.17 Memory usage for random 4-bit DFAs of 10 states .................. 64
7.18 Memory usage for random 4-bit DFAs of 100 states ................. 65
7.19 Memory usage for random 1-bit DFAs of 100 states, 70% density .... 65
7.20 Memory usage for random 1-bit DFAs of 1000 states, 70% density ... 66
7.21 Memory usage for random 2-bit DFAs of 100 states, 70% density .... 67
Summary

Finite state automata are used in a large number of applications in computing science. Therefore, any transformation operations that improve the performance of these automata are potentially useful. In this thesis we present two new, related transformation operations on automata, called stretching and jamming.

Automata are used to process strings with the use of a transition diagram. By stretching an automaton, every single transition in the transition diagram is split into two or more sequential transitions. If the transition diagram is implemented with a transition table, this can reduce the memory usage that is needed to store the transition diagram. Stretching is useful if the number of transitions in the transition table is low.

Jamming is the inverse operation of stretching. We can jam an automaton by transforming 2 or more sequential transitions into a single transition. Jamming reduces the string processing time. It is useful if the alphabet size of the automaton is small.
Chapter 0

Introduction

Automata theory forms an important part of theoretical computing science since its beginnings. It also forms the basis for a wide variety of practical applications like spell-checking, network intrusion detection and pattern matching. Compilers are an example of a classical application that uses automata theory.

In a compiler, a lexical analyzer is used to read input characters and to produce as output a sequence of tokens that the parser uses for syntax analysis [ASU86, p. 84]. Since the process of lexical analysis occupies a reasonable portion of the compiler’s time, the lexical analyzer should minimize the number of operations it performs per input character [ASU86, p. 144].

The lexical analyzer uses finite automata to recognize languages. This finite automaton uses a transition function to process strings but there are different ways to implement this transition function. The easiest and fastest way is to use a transition table in which there is a row for each state and a column for each input symbol. Unfortunately, this representation can take up a lot of space [ASU86, p. 114].

Thus, transformations on automata that increase their performance in terms of memory consumption or string recognition time are potentially useful (see for example [Wat95]).

We propose two transformations on automata: stretching and jamming. Under certain conditions, these transformations will produce more efficient automata in terms of memory consumption and string recognition time.

Given a deterministic finite automaton (DFA), we can stretch it by transforming each single transition into two or more sequential transitions, thereby introducing additional intermediate states. For example, an ASCII DFA can be stretched by transforming each single ASCII (8-bit) character transition into two transitions, each of 4-bit characters. Jamming is the inverse transformation, in which two successive transitions (based on, for example, input characters represented in 8-bits) are transformed into a single transition (in this example based on an input character represented in 16-bits). Of course, the same transformations can be used on a nondeterministic finite automaton (NFA).

In this thesis, we will study the stretching and jamming transformations and their effect on automata. In particular, we will investigate in which cases transforming an automaton has a positive effect on its performance. Therefore, the aim of this thesis is to determine the usefulness of the stretching and jamming transformations.

The material in this thesis is based on both theoretical results obtained by studying the transformations, and practical results obtained by benchmarking. We will start with the preliminaries in the next section to introduce the notations we use. Readers who are familiar
with automata theory might skip this section at first.

After that, in chapter 2, we introduce formal definitions of stretching and jamming. We will present a formal discussion of our main application of stretching and jamming, called bit-level stretching and jamming, in chapter 3.

The algorithms we developed are discussed in two separate chapters. In chapter 4, we give algorithms that are based on the formal definitions and formal discussion of our transformations. The algorithms are presented in an abstract way in the guarded command language (GCL). In the following chapter, several improvements on our algorithms are discussed. Readers who are only interested in the final algorithms can skip to this last chapter immediately. Others, who are interested in the successive development steps of our algorithms are recommended to read both chapter 4 and 5.

We implemented our algorithms in the C++ programming language and this implementation is discussed in chapter 6. We used the implementation to run several benchmarking tests to obtain practical results for both stretching and jamming. Readers who are primarily interested in the performance of stretching and jamming in practical situations are referred to chapter 7.

The main conclusions are presented in the last chapter, together with some indications for future work.
Chapter 1

Preliminaries

In this section we present the basic notions and notations used in this paper. Most of the notations used are standard, (see for example [HMU01]), but a few new notations are introduced.

A deterministic finite automaton, DFA, is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where $Q$ is a finite set of states, $\Sigma$ is the alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the (partial) transition function, $q_0$ is the initial state and $F$ is a subset of $Q$ whose elements are final states. $|Q|$ is the number of states and $|\Sigma|$ is the number of elements in the alphabet, or alphabet size.

The $n$-closure of an alphabet is the set of all symbols that consist of concatenating $n$ symbols from $\Sigma$. $\Sigma^+$ is the plus-closure of the alphabet, the set of symbols obtained by concatenating one or more symbols from $\Sigma$. We use the special alphabet $\mathbb{B} = \{0, 1\}$, the single bit alphabet. The $n$-closure of this alphabet allows us to define an alphabet of $n$ bits: $\mathbb{B}^n$.

$|Q||\Sigma|$ is the theoretical transition table size. Note that since cells represent states, the minimum cell size is determined by the minimum space requirements to represent a state, which is in turn determined by the total number of states. Although stretching and jamming will change the number of states in an automaton we will assume that the transition table cell size does not change in either transformation. We expect that in most cases the practical effects of this assumption are unlikely to be significant. We will verify this assumption in chapter 7, which contains the practical results of our transformations.

A transition in a DFA $M$ from $p$ to $q$ with label $a$ will be denoted by $(p, a, q)$ where $(p, a, q) \in Q \times \Sigma \times Q$ and $q = \delta(p, a)$. We will also use the notation $((p, a), q) \in \delta$.

A path of length $k$ in a DFA $M$ is a sequence $\langle (r_0, a_0, r_1), \ldots, (r_{k-1}, a_{k-1}, r_k) \rangle$, where $(r_i, a_i, r_{i+1}) \in Q \times \Sigma \times Q$ and $r_{i+1} = \delta(r_i, a_i)$ for $0 \leq i < k$. The word $a_0a_1\cdots a_{k-1} \in \Sigma^k$ is the label of the path.

The extended transition function of a DFA $M$, $\hat{\delta} : Q \times \Sigma^+ \rightarrow Q$, is defined so that $\hat{\delta}(r, w) = r_j$ if there is a path from $r_i$ to $r_j$, labeled $w$.

A nondeterministic finite automaton, NFA, is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, defined in the same way as a DFA, with the following exception: $\hat{\delta} : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ is the transition function. Note that $\mathcal{P}(Q)$ is the powerset of $Q$, the set consisting of all possible subsets of $Q$, this is sometimes written as $2^Q$ in the literature.

Note that in most literature there is another difference between NFAs and DFAs. Usually, the definition of an NFA includes the possibility of empty transitions, or $\epsilon$-transitions. An $\epsilon$-transition can be made without the input of a symbol. Because we don’t need it, and because
we can construct an equivalent NFA without $\epsilon$-transitions for every NFA with $\epsilon$-transitions, we can leave it out of our definition without any loss of generality.

A transition in an NFA $M$ from $p$ to $q$ with label $a$ will also be denoted by $(p, a, q)$ where $(p, a, q) \in Q \times \Sigma \times Q$ and $q \in \delta(p, a)$. In this case we will also use the notation $((p, a), q) \in \delta$. Strictly speaking this notation is not correct because, as said before, $\delta$ has signature $Q \times \Sigma \rightarrow \mathcal{P}(Q)$ in the case of an NFA. However, we take this liberty with the notation in the interests of conciseness.

A path of length $k$ in an NFA $M$ is a sequence $\langle (r_0, a_0, r_1), \ldots, (r_{k-1}, a_{k-1}, r_k) \rangle$, where $(r_i, a_i, r_{i+1}) \in Q \times \Sigma \times Q$ and $r_{i+1} \in \delta(r_i, a_i)$ for $0 \leq i < k$.

In an NFA $M$, the extended transition function, $\hat{\delta} : Q \times \Sigma^+ \rightarrow \mathcal{P}(Q)$, is also defined so that $r_j \in \hat{\delta}(r_i, w)$ iff there is a path from $r_i$ to $r_j$, labeled $w$.

If there is a path of length $k$ from $r_0$ to $r_k$ with label $w$ in a DFA or an NFA we will denote this by $\langle r_0, w, r_k \rangle$. Note that this notation is shorthand for $\{((r_0, a_0), r_1), \ldots, ((r_{k-1}, a_{k-1}), r_k)\}$, where $a_0 \ldots a_k = w$. This allows us to use the notation $\langle r_0, w, r_k \rangle \subseteq \delta$. 


Chapter 2

Definitions of Stretching and Jamming

2.0 General Definitions

In this section we give formal definitions of stretching and jamming. One way in which we can stretch an automaton is by transforming each transition into $k$ sequential transitions, and $k - 1$ additional intermediate states. This stretching operation on a single transition is pictured in figure 2.0.

![Figure 2.0: Stretching transition $(p, a, q)$ into $k$ sequential transitions.](image)

Jamming is the inverse transformation, in which $k$ sequential transitions are transformed into a single transition, removing the redundant intermediate states. In figure 2.0 this can be seen as performing the stretching transformation in opposite direction.

We can stretch NFAs as well as DFAs. We will define the stretching transformation on NFAs, and the result of a transformation will also be an NFA. Because DFAs are a subset of NFAs, the stretching of DFAs is automatically defined. If we stretch a DFA, in some cases the resulting automaton may have more than one transition with the same label from a given state and therefore the result of stretching a DFA might be an NFA. In chapter 5 we will present a way to stretch DFAs to DFAs, but for now we will investigate the more general case.

If NFA $FA_0$ can be stretched into NFA $FA_1$, we call $FA_1$ a ‘stretch’ of $FA_0$. The set of states of $FA_1$ consists of a subset $S_1$ of ‘original’ states and a subset $I$ of newly introduced additional intermediate states. Definition 2.0 formally describes a general stretching transformation.
Firstly, there is an injection from the alphabet of FA$_0$, $\Sigma_0$ to $\Sigma_1^+$, the plus-closure of the alphabet of FA$_1$ (property 2.0.0). Second, there is a one-to-one relation between the start states of FA$_0$ and FA$_1$ (property 2.0.1). The third property (2.0.2) defines a one-to-one relationship between the final states of both automata. Property 2.0.3 states that for every transition from state $p$ to $q$ with label $a$ in FA$_0$ there exists a path from $\varphi(p)$ to $\varphi(q)$ with label $\tau(a)$ in FA$_1$, which travels from a state in $S_1$ via a number of intermediate states to another state in $S_1$. The inverse of this property is also true, therefore property 2.0.4 also holds.

**Definition 2.0.** Let FA$_0 = (S_0, \Sigma_0, \delta_0, q_0, F_0)$ be an NFA, let FA$_1 = (S_1 \cup I, \Sigma_1, \delta_1, q_1, F_1)$ be an NFA, and $|F_0| = |F_1|$. FA$_1$ is a stretch of FA$_0$ iff:

- There is an injection $\tau : \Sigma_0 \to \Sigma_1^+$, thus:
  $$(\forall a_0, a_1 : a_0, a_1 \in \Sigma_0 : \tau(a_0) = \tau(a_1) \Rightarrow a_0 = a_1) \quad (2.0.0)$$
- There is a bijection $\varphi : S_0 \leftrightarrow S_1$, with the following properties:
  - $\varphi(q_0) = q_1$ \quad (2.0.1)
  - $$(\forall f_0 \in F_0 : (\exists f_1 \in F_1 : \varphi(f_0) = f_1)) \quad (2.0.2)$$
  - $$(\forall ((p, a), q) : ((p, a), q) \in \delta_0 : (\exists k, (r_0, w, r_k) : (r_0, w, r_k) \subseteq \delta_1 : \varphi(p) = r_0 \land \varphi(q) = r_k \land \tau(a) = w)) \quad (2.0.3)$$
  - $$(\forall k, (r_0, w, r_k) : (r_0, w, r_k) \subseteq \delta_1 : (((\varphi(r_0), \tau^{-1}(w)), \varphi(r_k)) \in \delta_0)) \quad (2.0.4)$$

where $p, q \in S_0$, $a \in \Sigma_0$, $r_0, r_k \in S_1$, $r_1, \ldots, r_{k-1} \in I$ and $w \in \Sigma_1^k$ for $k \geq 1$.

We define jamming as the inverse transformation of stretching. Because of symmetry, if we jam certain NFAs the result will be a DFA.

If NFA FA$_0$ is jammed into NFA FA$_1$ (FA$_1$ is a ‘jam’ of FA$_0$) then FA$_0$ is a stretch of FA$_1$. The set of states of FA$_0$ consists of a subset $S_0$ of ‘original’ states and a subset $R$ of redundant intermediate states. These redundant intermediate states will be removed by the jamming transformation.

**Definition 2.1.** Let FA$_0 = (S_0 \cup R, \Sigma_0, \delta_0, q_0, F_0)$ be an NFA, and let FA$_1 = (S_1, \Sigma_1, \delta_1, q_1, F_1)$ be an NFA. FA$_1$ is a jam of FA$_0$ iff FA$_0$ is a stretch of FA$_1$, with $R$ being the set of additional intermediate states, resulting from stretching.

2.1 Stretching and Jamming by a Factor $f$

In the previous section we presented definition 2.0. This definition is very general, every single transition can be stretched into a different number of sequential transitions, according to properties 2.0.3 and 2.0.4.

In the next chapter our major application of stretching and jamming will be introduced. To be able to use the definitions in that application, we need to make a restriction on the
previous definition. In the definition below we only allow transitions to be stretched into a fixed number of sequential transitions.

Therefore, we introduce the factor \( f \) in stretching and jamming. If NFA \( FA_0 \) is stretched by a factor \( f \) into NFA \( FA_1 \) (\( FA_1 \) is an ‘f-stretch’ of \( FA_0 \)) then there is a one-to-one relation between an element from the alphabet of \( FA_0 \) and the \( f \)-closure of the alphabet of \( FA_1 \). Furthermore, for each transition in \( FA_0 \) there are exactly \( f \) sequential transitions in \( FA_1 \).

The formal differences between the new definition and the previous definition are expressed in properties 2.2.1 and 2.2.4/2.2.5. In this definition, the relation \( \tau \) is not only an injection, but because of property 2.2.1 also a surjection, and therefore a bijection. Also, because of property 2.2.4, for every transition \((p, a, q)\) in the original NFA, there is a path of exactly length \( f \) from \( \varphi(p) \) to \( \varphi(q) \) with label \( \tau(a) \). Property 2.2.5 states that the inverse is also true.

**Definition 2.2.** Let \( FA_0 = (S_0, \Sigma_0, \delta_0, q_0, F_0) \) be an NFA, let \( FA_1 = (S_1 \cup I, \Sigma_1, \delta_1, q_1, F_1) \) be an NFA, and \( |F_0| = |F_1| \). \( FA_1 \) is an ‘f-stretch’ of \( FA_0 \) iff:

- \( FA_1 \) is a stretch of \( FA_0 \), such that the injection \( \tau \) specializes to a bijection \( \tau : \Sigma_0 \leftrightarrow \Sigma_1^f \), thus:
  \[ (\forall a_0, a_1 : a_0, a_1 \in \Sigma_0 : \tau(a_0) = \tau(a_1) \Rightarrow a_0 = a_1) \quad (2.2.0) \]
  \[ (\forall w \in \Sigma_1^f : (\exists a \in \Sigma_0 : \tau(a) = w)) \quad (2.2.1) \]
- The bijection \( \varphi : S_0 \leftrightarrow S_1 \), is characterized by:
  \[ \varphi(q_0) = q_1 \quad (2.2.2) \]
  \[ (\forall f_0 \in F_0 : (\exists f_1 \in F_1 : \varphi(f_0) = f_1)) \quad (2.2.3) \]
  \[ (\forall ((p, a), q) : ((p, a), q) \in \delta_0 : (\exists (r_0, w, r_f) : (r_0, w, r_f) \subseteq \delta_1 : \varphi(r_0) = p \land \varphi(r_f) = q \land \tau(a) = w)) \quad (2.2.4) \]
  \[ (\forall (r_0, w, r_f) : (r_0, w, r_f) \subseteq \delta_1 : (((\varphi(r_0), \tau^{-1}(w)), \varphi(r_f)) \in \delta_0)) \quad (2.2.5) \]

where \( p, q \in S_0, a \in \Sigma_0, r_0, r_f \in S_1, r_1, \ldots, r_{f-1} \in I \) and \( w \in \Sigma_1^f \) for \( 0 \leq i < f \).

Jamming by a factor \( f \) is defined analogously to stretching by a factor \( f \). Note that, by definition, jamming by a factor \( f \) is not always possible. NFA \( FA_0 \) can only be jammed if there exists an NFA \( FA_0 \) which can be stretched into \( FA_0 \). See figure 5.1 in chapter 5 for an example of an NFA that can not be jammed according to our definition. In that same chapter we propose a solution for this problem.

**Definition 2.3.** Let \( FA_0 = (S_0 \cup R, \Sigma_0, \delta_0, q_0, F_0) \) be an NFA, and let \( FA_1 = (S_1, \Sigma_1, \delta_1, q_1, F_1) \) be an NFA. \( FA_1 \) is an ‘f-jam’ of \( FA_0 \) iff \( FA_0 \) is an ‘f-stretch’ of \( FA_1 \). The set \( R \) of \( FA_0 \) is the set of additional intermediate states, resulting from stretching.

To illustrate these definitions we give an example:
Example 2.4. The graph in figure 2.1 represents the DFA 
\( FA_0 = (\{q, r, s\}, \{a, b, c, d\}, \delta_1, q, \{s\}) \). DFA \( FA_0 \) can be stretched by a factor 2 into NFA \( FA_1 \) of figure 2.2. NFA \( FA_1 = (S_2 \cup I, \{e, f, g, h, i, j, k\}, \delta_2, t, \{y\}) \), with \( S_2 = \{t, w, y\} \) and \( I \), the set of additional intermediate states, is \( \{u, v, x, z\} \). Injection \( \tau \) and bijection \( \varphi \) are shown in figure 2.3 and 2.4 respectively.

In this example we can see why stretching a DFA can result in an NFA. Because we chose \( \tau(b) = gh \) and \( \tau(c) = gi \), there are two outgoing transitions with label \( g \) from state \( w \). Therefore \( FA_1 \) is an NFA. 

\[
\begin{align*}
\tau(a) &= ef \\
\tau(b) &= gh \\
\tau(c) &= gi \\
\tau(d) &= jk
\end{align*}
\]
Chapter 3

Bit-level Stretching and Jamming

In this section we present an application of stretching and jamming. We look at stretching and jamming on a bit-level. We will only consider automata in which each element of the alphabet is a bit string. An \textit{n-bit automaton} is an automaton whose alphabet consists of all the \(2^n\) bit strings of length \(n\).

**Proposition 3.0.** Let \(f\) be a factor of \(n\). Then we can \(f\)-stretch the \(n\)-bit DFA \(F \! A_0\) into NFA \(F \! A_1\) in the following way:

- \(F \! A_1\) is an \(\frac{n}{f}\) – bit NFA.
- There is a bijection between \(\Sigma_0\) and \(\Sigma_1^f\) ie. for every bit string of length \(n\) in \(\Sigma_0\) there is a sequence of \(f\) bit strings of length \(\frac{n}{f}\) in \(\Sigma_1^f\) and vice versa.
- For every transition in \(F \! A_0\) there are \(f\) sequential transitions in \(F \! A_1\), obeying the above bijection between \(\Sigma_0\) and \(\Sigma_1^f\) for the labels of the transitions.

Of course, this specialization of stretching is only allowed if \(n\) is divisible by \(f\). In that case we call the DFA ‘\(f\)-stretchable’. Again, jamming is the opposite transformation: if an \(n\)-bit NFA is ‘\(f\)-jammable’, the resulting automaton is an \(nf\) – bit DFA.

Example 3.1. To illustrate the stretching of \(n\)-bit automata we give an example. The 2-bit DFA \(F \! A_0\) of figure 3.0 can be stretched by a factor 2 into the 1-bit NFA \(F \! A_1\) of figure 3.2. Also, the 1-bit NFA \(F \! A_1\) can be jammed into the 2-bit DFA \(F \! A_0\). Furthermore, NFA \(F \! A_1\) can be determinized into DFA \(F \! A_2\) of figure 3.4.
Note that in the previous example, we stretch a transition by using the most significant bit first. For example, we stretch transition \((a, 01, c)\) into \((a, 0, i_1)\) and \((i_1, 1, c)\), taking the 0 first and then the 1. In practice, we will usually use the least significant bit first, because that is the most natural way to process a bit string. It can be done as presented here in practice too, however. This will only involve a minor change in the stretch algorithms we present.

### 3.0 Theoretical Results for Stretching

In this section we will prove a number of propositions about stretching and jamming. From these propositions we can draw conclusions about when stretching or jamming would be useful in terms of memory consumption and string recognition time.

The first four propositions will deal with bit-level stretching. For those first four propositions we assume that \(n\)-bit DFA \(M = (Q, \Sigma, \delta, q_0, F)\) with the number of states \(r = |Q|\), the alphabet size \(s = |\Sigma|\) and the number of transitions \(t\), is stretched by a factor \(f\) into \(\frac{n}{f}\)-bit NFA \(M' = (Q', \Sigma', \delta', q'_0, F')\), with \(Q' = S \cup I\), \(r' = |Q'|\), \(s' = |\Sigma'|\) and \(t'\) the number of transitions. The number of additional intermediate states in NFA \(M'\) is \(I = r' - r\). The transition table sizes are respectively \(rs\) and \(r's'\).

The following propositions hold:

**Proposition 3.2.** \(s' = \sqrt[fs]{s}\)

**Proof.** If \(M\) is an \(n\)-bit DFA, then the alphabet size, \(s\), is \(2^n\). If \(M\) is stretched by a factor \(f\) into \(M'\), \(M'\) is an \(\frac{n}{f}\)-bit NFA, so the alphabet size, \(s'\), is \(2^{\frac{n}{f}} = \sqrt[fs]{s}\).
Proposition 3.3. \( r' = t(f - 1) + r \)

Proof. Stretching by a factor \( f \) introduces \( f - 1 \) additional intermediate states, for each single transition in the original DFA. Therefore \( r' \) is equal to the additional intermediate states, \( t(f - 1) \), plus the number of states in the original DFA, \( r \).

Proposition 3.4. \( r \leq r' \leq rs(f - 1) + r \)

Proof. From proposition 3.3 we know that if DFA \( M \), with \( t \) transitions, is stretched by a factor \( f \) into \( M' \), \( r' = t(f - 1) + r \). The number of transitions is at least 0 and at most \( rs \). Therefore, \( r \leq r' \leq rs(f - 1) + r \).

Proposition 3.5. Let \( z = \frac{r(s - \sqrt{s})}{\sqrt{s}(f - 1)} \).

\[
\begin{align*}
& t < z \iff r's' < rs \\
& t = z \iff r's' = rs \\
& t > z \iff r's' > rs
\end{align*}
\]

Proof. From proposition 3.2 we know that if DFA \( M \) is stretched by a factor \( f \) to \( M' \), \( s' = \sqrt{s} \). From proposition 3.3 we know that if DFA \( M \), with \( t \) transitions, is stretched by a factor \( f \) to \( M' \), \( r' = t(f - 1) + r \). Thus, \( r's' = (t(f - 1) + r) \sqrt{s} \).

If \( t = z = \frac{r(s - \sqrt{s})}{\sqrt{s}(f - 1)} \) then:

\[
r's' = \left( \frac{r(s - \sqrt{s})}{\sqrt{s}(f - 1)} \right) (f - 1) + r \sqrt{s} = rs
\]

The inequalities follow directly by similar reasoning.

From proposition 3.5 we can conclude that bit-stretching a DFA reduces the transition table size when \( t < z \). Therefore it can reduce the amount of memory needed for a transition table representation.

The transition density of an automaton is the number of transitions divided by the transition table size (the maximum number of possible transitions), \( \frac{t}{rs} \). The density needed to get an equal transition table size after stretching a DFA (the break-even point) is \( \frac{z}{rs} = \frac{1}{1 - f} \). This value is not dependent on the number of states, \( r \). In figure 3.6, \( \frac{z}{rs} \) is set out against the alphabet size \( s \) for different values of \( f \). From these graphs we conclude that the transition density has to be very low to obtain a smaller transition table size by stretching.

### 3.1 Theoretical Results for Jamming

The next three propositions will deal with bit-level jamming. For these propositions we assume that \( n \)-bit NFA \( M = (Q, \Sigma, \delta, q_0, F) \) with \( Q = S \cup R \), the number of states \( r = |Q| \), the alphabet size \( s = |\Sigma| \) and the number of transitions \( t \), is jammed by a factor \( f \) into \( nf \)-bit DFA \( M' = (Q', \Sigma', s', q'_0, F') \), with \( r' = |Q'| \), \( s' = |\Sigma'| \) and \( t' \) the number of transitions. The number of redundant intermediate states in NFA \( M \) is \( R = r - r' \). The transition table sizes are respectively \( rs \) and \( r's' \).

Proposition 3.6. \( s' = s'^f \)
Proof. If $M$ is an $n$-bit NFA, then the alphabet size, $s$, is $2^n$. If $M$ is jammed by a factor $f$ into $M'$, $M'$ is an $nf$-bit DFA, so the alphabet size, $s'$, is $2^{nf} = (2^n)^f = s^f$. □

**Proposition 3.7.** $0 \leq r' \leq r$

Proof. Every automaton has at least 0 states, thus $r' \geq 0$. Jamming removes all redundant intermediate states so $r' \leq r$. □

**Proposition 3.8.** Let $z = r - \frac{r}{s^f r}$, and $r - r'$ the number of redundant intermediate states.

$r - r' > z \iff r's' < rs$
$r - r' = z \iff r's' = rs$
$r - r' < z \iff r's' > rs$

Proof. From proposition 3.6 we know that if NFA $M$ is jammed by a factor $f$ to into $M'$ then $s' = s^f$. If $M$ is jammed into $M'$ then $r - r'$ states are removed. So if $r - r' = r - \frac{r}{s^f r}$ then $r' = r - (r - \frac{r}{s^f r})$. Therefore, $r's' = \frac{r}{s^f r} s^f = rs$. Again, the inequalities follow directly by similar reasoning. □
From these propositions we can conclude that bit-jamming an NFA can potentially reduce the number of states and the number of transitions. Therefore it can reduce the average time needed to recognize a string.

The redundant state density of an automaton is the number of redundant intermediate states divided by the total number of states, \( \frac{r - r'}{r} \). The density needed to get an equal transition table size after jamming an NFA is \( \frac{z}{r} = 1 - s^{1-f} \). This value is not dependent on the number of states, \( r \).

In figure 3.7, \( \frac{z}{r} \) is set out against the alphabet size \( s \) for different values of \( f \). From these graphs we can conclude that the redundant state density has to be very high to obtain a smaller transition table size by jamming. While these theoretical results indicate that jamming will often require a larger transition table, we expect that it will reduce the average string recognition time. However, this has to be examined in practice.

**Example 3.9.** Figure 3.8 visually represents how the dimensions of the transition tables change, by virtue of propositions 3.2, 3.4, 3.6 and 3.7.

This concludes our overview of bit-level stretching and jamming. We end with a few notes. In this chapter, and in the rest of this thesis, we will only consider stretching or jamming the complete transition table. As we saw in this section, stretching is useful if the transition density is low. Therefore, it might be interesting to stretch only certain parts of the transition table where the transition density is low. For jamming we can argue that if a certain part of the transition table contains many redundant states, jamming this part only might be interesting. We call this approach local stretching and jamming but leave it as future work.

Furthermore, we will only consider transition tables that can be implemented with a regular matrix that has a row for each state and a column for each input symbol. Of course, there are cases where this does not apply. For example, sometimes character classes or sparse matrices are used for implementing the transition table. We will not discuss these situations in this thesis.

As a general guideline however, if a certain transition table implementation leads to a smaller transition table density it will have a positive effect on stretching. If it leads to a higher transition density it will have a negative effect. Similar reasoning applies for jamming.
In this chapter we present three algorithms, one for stretching, one to determine if an NFA is jammable and one for jamming. First, we need some definitions. We use \( \text{mod} \) as the operator for the remainder of integer division.

In our stretch algorithm we conceive of alphabet elements as strings of subelements (typically bit substrings). If alphabet element \( a \in \Sigma \) has length \( |a| \) then we number the subelements \( a.0, \ldots, a.(|a| - 1) \). Thus, if \( a = 0111 \) then \( a.0 = 0, a.1 = 1, a.2 = 1 \) and \( a.3 = 1 \).

We use square brackets to denote a substring of an alphabet element. The length of the substring depends on the factor \( f \) used in stretching. For \( a \in \Sigma \), \( a[i] = a.(i \mod |a| \cdot f) \). So for example, if \( \text{FA}_0 \) is stretched by a factor 2 and \( a = 0111 \) (so \( |a| = 4 \)), then \( a[0] = 01 \) and \( a[1] = 11 \). If \( \text{FA}_0 \) is stretched by a factor 4 then \( a[0] = 0, a[1] = 1, a[2] = 1 \) and \( a[3] = 1 \).

We use Dijkstra’s guarded command language (GCL) to present our algorithms. For an explanation of GCL, see [Dij76]. The main reason for using GCL is that our algorithms can be presented efficiently in this language. Furthermore, because we choose GCL as presentation language we can choose any imperative programming language for the implementation of the algorithms.

We make one extension to GCL for the alternative construct. By definition, the alternative construct gets stuck if none if its guards evaluate to true. Therefore, if we want statement \( S \) to be executed if guard \( B \) is true and nothing executed otherwise, this is notated as follows:

\[
\begin{align*}
\text{if } B \rightarrow & S \\
\text{else if } \neg B \rightarrow & \text{skip} \\
\text{fi}
\end{align*}
\]

We use the following alternative notation for this case:

\[
\text{as } B \rightarrow S \text{ sa}
\]

This way, our algorithms are easier to read.

### 4.0 Stretch Algorithm

In this section we present a stretch algorithm, algorithm 4.1. This algorithm takes as input NFA \( \text{FA}_0 \) and a factor \( f \). The output of the algorithm is NFA \( \text{FA}_1 \), an \( f \)-stretch of \( \text{FA}_0 \).
We can stretch an NFA (or DFA) by a factor \( f \) by stretching all transitions into \( f \) sequential transitions. We initialize \( F A_1 \) with the set of states, start state and final states of \( F A_0 \). Initially, the transition function \( \delta_1 \) of \( F A_1 \) is empty.

Then, for every transition \((p, a, q)\) in \( F A_0 \) we add the new intermediate states \( i_{pq}^0, \ldots, i_{pq}^{f-2} \) to \( F A_1 \). After that, transitions \((p, a[0], i_{pq}^0), (i_{pq}^0, a[1], i_{pq}^1), \ldots, (i_{pq}^{f-2}, a[f-1], q)\) are added to \( \delta_1 \).

Because we implement the transition function using a transition table representation, it is easy to iterate over all transitions. We simply can iterate over all the cells in the transition table representation.

An advantage of iterating over the cells in the transition table is that not all states have to be reachable from the start state. This is not the most efficient implementation in terms of running time though. In the next chapter we will present some improvements to this algorithm.

**Proposition 4.0.** Let \( F A_0 = (S_0, \Sigma_0, \delta_0, q_0, F_0) \) be an \( n \)-bit NFA. Algorithm 4.1 will stretch \( F A_0 \) by a factor \( f \) into \( \frac{n}{f} \) - bit NFA \( F A_1 = (S_1, \Sigma_1, \delta_1, q_1, F_1) \).

**Algorithm 4.1 STRETCH\((F A_0, F A_1, f)\):**

\[
\begin{align*}
\text{Pre : } & n \in \mathbb{Z}^+ \land f \in \mathbb{Z}^+ \land n \mod f = 0 \\
\text{Post : } & F A_1 \text{ is an } f\text{-stretch of } F A_0 \\
 & \left[ S_1 := S_0, q_1 := q_0, F_1 := F_0, \Sigma_1 := \mathbb{B}^{\frac{n}{f}}, \delta_1 := \emptyset \right] \\
 & \text{for all } ((p, a), q) : ((p, a), q) \in \delta_0 \to \\
 & \hspace{1cm} \begin{array}{l}
 s_{pq}^0 := p \\
 I := I \cup \{ s_{pq}^1, \ldots, s_{pq}^{f-1} \} \\
 s_{pq}^f := q \\
 \text{for } i := 0 \to f - 1 \to \delta_1 := \delta_1 \cup \{ (s_{i, a[i]}, s_{i+1}) \} \\
 \text{rof}
\end{array}
\end{align*}
\]

**Proof of proposition 4.0.**

We need to prove that algorithm 4.1 will stretch \( F A_0 \) according to the conditions in definition 2.2.

- We have to prove there exists a bijection \( \tau : \Sigma_0 \leftrightarrow \Sigma_1^f \). Because \( \Sigma_1^f = (\mathbb{B}^{\frac{n}{f}})^f = \mathbb{B}^n = \Sigma_0 \), there is a trivial bijection between the two sets. Intuitively, we can see that every bit string of length \( n \) can be constructed from a unique sequence of \( f \) bit strings of length \( \frac{n}{f} \).

- After the algorithm is executed, \( S_0 = S_1, q_0 = q_1 \) and \( F_0 = F_1 \), therefore there is an obvious bijection \( \varphi : S_0 \leftrightarrow S_1 \). For each transition \(((p, a), q) \in \delta_0\), the path \( \{ ((s_{pq}^0, a[0]), s_{pq}^1), (s_{pq}^1, a[1]), s_{pq}^2), \ldots, ((s_{pq}^{f-1}, a[f-1]), s_{pq}^f) \} \) is added to \( \delta_1 \). In this path, \( s_{pq}^0 = p, s_{pq}^f = q, \) and \( s_{pq}^1, \ldots, s_{pq}^{f-1} \) are new states that are added to set \( I \). Because
\[ \varphi(p) = p \text{ and } \varphi(q) = q \text{ and } \tau(a) = a[0] \ldots a[f - 1] = a, \text{ property 2.2.4 holds.} \]

And because initially \( \delta_1 \) is empty, property 2.2.5 also holds.

In the previous chapter, we indicated that if we stretch a transition we can either use the most significant or the least significant bit first. In the previous algorithm we use the least significant bit first. If we want to use the most significant bit first, we have to use the last part of symbol \( a \) for the first transition we add. For the second sequential transition, we have to use the second last part, and so on.

Therefore, if we want to use the most significant bit first, we can simply use \( a[f - 1 - i] \) as the new label for each stretched transition instead of \( a[i] \). We can do the same for the other algorithms we will introduce later, so we will not discuss it further in the development of our algorithms. In chapter 7 we will present some benchmarking results that will show the difference between stretching the most significant or least significant bit first.

### 4.1 Jam Algorithms

In this section we will present our first jam algorithms. In jamming, we only consider NFAs in which all states are on a path from the start state and all states are on a path to at least one of the final states. In chapter 2 we saw that, by definition, not every NFA is jammable so first we will give the properties that are needed for an NFA to be jammable by a factor \( f \). After that we will present an algorithm to determine if an NFA is jammable by a factor 2. This algorithm can be generalized to an algorithm that determines if an NFA is jammable by a factor \( f \).

**Proposition 4.2.** Let \( FA_0 = (S_0, \Sigma_0 = B^n, \delta_0, q_0, F_0) \) be an \( n \)-bit NFA. \( FA_0 \) is jammable by a factor \( f \) iff there exists a division of \( S_0 \) into mutually disjoint sets \( T_0, T_1, \ldots, T_{f-1} \) with the following properties:

1. \( q_0 \in T_0 \) \hfill (4.2.0)
2. \( F_0 \subseteq T_0 \) \hfill (4.2.1)
3. \( \forall p, a, q : ((p, a), q) \in \delta_0 : p \in T_i \Rightarrow q \in T_{i+1 \mod f} \) \hfill (4.2.2)

**Proof of proposition 4.2.**

‘if’: We have to prove that if there exists a division of \( S_0 \) into disjoint sets \( T_0, T_1, \ldots, T_{f-1} \) with properties 4.2.0, 4.2.1 and 4.2.2, \( FA_0 \) is jammable by a factor \( f \). In other words, we have to prove that there exists an NFA \( FA_1 = (S_1, \Sigma_1 = B^{nf}, \delta_1, q_1, F_1) \) which can be stretched into NFA \( FA_0 \) according to definition 2.2.

The bijection \( \tau : \Sigma_0 \leftrightarrow \Sigma_1 \) is obtained automatically, because \( \Sigma_1 = (B^n)^f = B^{nf} = \Sigma_0 \).

We can partition \( S_0 \) into sets \( T_0 \) and \( I \), with \( I = T_1 \cup T_2 \cup \ldots \cup T_{f-1} \). Now, if we choose \( S_1 = T_0 \), we can have a trivial bijection \( \varphi : T_0 \leftrightarrow S_1 \). Because of properties 4.2.0 and 4.2.1, \( q_0 \) and \( F_0 \) are included in this bijection, so properties 2.2.2 and 2.2.3 hold.

Furthermore, because of property 4.2.2, we know that every path from a state \( p \in T_0 \) to a state \( q \in T_0 \) has length \( f \). If this path has label \( w \), we can have a transition
\((\varphi(p), \tau^{-1}(w)), \varphi(q)\) \in \delta_1\), that can be stretched into that path \(\langle p, w, q \rangle\). This can be done for all paths, and therefore all transitions, so property 2.2.5 holds. Because of symmetry, property 2.2.4 also holds.

\textbf{Only if}: We have to prove that if \(F A_0\) is jammable by a factor \(f\), there exists a division of \(S_0\) into disjoint sets \(T_0, T_1, \ldots, T_{f-1}\) with the properties of proposition 4.2. In other words, we have to prove that if we can stretch \(n\)-bit \(F A_1 = (S_1, \Sigma_1 = B^n, \delta_1, q_1, F_1)\) into \(F A_0 = (S_0 \cup I, \Sigma_0 = B^n, \delta_0, q_0, F_0)\) according to definition 2.2, there exists a division of \(S_0\) into \(f\) mutual disjoint sets so that all properties of proposition 4.2 hold for \(F A_0\).

If we stretch \(F A_1\), then there exists a bijection \(\varphi : U_0 \leftrightarrow S_1\), with \(S_0 = U_0 \cup I\). If we choose \(U_0 = T_0\), and \(I = T_1 \cup T_2 \cup \ldots \cup T_{f-1}\) properties 4.2.0 and 4.2.1 hold trivially.

Because of property 2.2.4, for all paths \((p, a, q) \in \delta_0\) there exists a path \(\langle r_0, w, r_f \rangle\) in \(\delta_1\), with \(r_0, r_f \in T_0\) and \(r_1, \ldots, r_{f-1} \in I\). Because this path has length exactly \(f\), and because \(r_1, \ldots, r_{f-1}\) are all newly introduced states, we can stretch \(F A_1\) in such a way that \(r_1 \in T_1, \ldots, r_{f-1} \in T_{f-1}\). Therefore property 4.2.2 holds.

\[\square\]

**Remark 4.3.** In graph theory, if the set \(V\) of vertices of graph \(G\) can be divided into disjoint sets \(W_0, W_1, \ldots, W_{k-1}\), \(G\) is called \(k\)-partite (bipartite in the case \(k = 2\)). Because NFAs (and DFAs) can be represented as a graph we can also speak about \(k\)- or bipartite NFAs (DFAs).

The following algorithms will traverse the NFA in a breadth-first manner. This way, every state is discovered only once and states at distance \(k\) from start state \(q_0\) are discovered before states at distance \(k+1\). To use this way of discovering transitions, all states must be reachable from the start state.

To be able to traverse an NFA in a breadth-first manner we have based the algorithms on the well known Breadth-First Search (BFS) algorithm. The BFS algorithm given below is a simplification of the BFS algorithm given in [CLR01]. A correctness proof can also be found in [CLR01].

The function \(\text{enqueue}(q, Q)\) inserts element \(q\) in queue \(Q\). The function \(\text{dequeue}(Q)\) returns the first element of queue \(Q\) and removes it.

**Proposition 4.4.** Let \(F A_0 = (S_0, \Sigma_0, \delta_0, q_0, F_0)\) be an NFA. Algorithm 4.5 will traverse \(F A_0\) in a breadth-first search manner.

**Algorithm 4.5 BFS(FA0):**

\[
\begin{array}{l}
\text{\| } Q, V := \emptyset, \emptyset \\
\text{;enqueue}(q_0, Q) \\
\text{;do } Q \neq \emptyset \rightarrow \\
\quad p := \text{dequeue}(Q) \\
\quad \text{for all } q, a : ((p, a), q) \in \delta_0 \rightarrow \\
\quad \text{as } q \notin V \rightarrow \\
\quad \qquad V := V \cup \{q\} \\
\quad \text{;enqueue}(q, Q) \\
\quad \text{;rof}
\end{array}
\]
Proof of proposition 4.4. A correctness proof can be found in [CLR01]. □

**Proposition 4.6.** Let $FA_0 = (S_0, \Sigma_0, \delta_0, q_0, F_0)$ be an $n$-bit NFA. Algorithm 4.7 will determine if $FA_0$ is jammable by a factor 2. □

**Algorithm 4.7 JAMMABLE($FA_0$):**

1. **Pre:** $n \in \mathbb{Z}^+ \land n \mod 2 = 0$
2. **Post:** The algorithm will return if $FA_0$ is jammable by a factor 2 or not

\[
\begin{align*}
Q & := \emptyset \\
E, O & := \emptyset, \emptyset \\
\text{enqueue}(q_0, Q) \\
\text{do } Q \neq \emptyset & \rightarrow \\
& \quad p := \text{dequeue}(Q) \\
& \quad \text{for all } q, a : ((p, a), q) \in \delta_0 \rightarrow \\
& \quad \quad \text{as } q \notin E \cup O \land p \in E \land q \notin F_0 \rightarrow O := O \cup \{q\}; \text{ enqueue}(q, Q) \\
& \quad \quad q \notin E \cup O \land p \in E \land q \in F_0 \rightarrow \text{return false} \\
& \quad \quad q \notin E \cup O \land p \in O \rightarrow E := E \cup \{q\}; \text{ enqueue}(q, Q) \\
& \quad \quad q \in E \land p \in E \rightarrow \text{return false} \\
& \quad \quad q \in O \land p \in O \rightarrow \text{return false} \\
& \quad \text{rof} \\
& \text{od} \\
& \text{;return true}
\end{align*}
\]

Proof of proposition 4.6.
The algorithm is based on the BFS algorithm 4.5. The BFS algorithm ensures that if a new state $q$ is found, the path traveled by the algorithm from $q_0$ to $q$ is the shortest path. The algorithm searches for all states $q$ that have an incoming transition from a found state $p$.

The algorithm divides all states in two subsets $E$ and $O$, corresponding to $T_0$ and $T_1$ respectively, from proposition 4.2. Set $E$ is the set of states that are on even distance from start state $q_0$, the states in set $O$ are on odd distance.

If $q \notin E \cup O$ then $q$ has not been found yet. This means that it can be added to the appropriate set $E$ or $O$ unless $p \in E$ and $q \in F_0$. If this is the case then $q$ must be added to $O$ which would violate property 4.2.1 of proposition 4.2.

If $q \in E$ or $q \in O$ then $q$ has already been found. If $p \in E$ then $q$ must be in $O$, otherwise there is a violation of property 4.2.2. Analogously, if $p \in O$ then $q$ must be in $E$. □

**Proposition 4.8.** Let $FA_0 = (S_0, \Sigma_0, \delta_0, q_0, F_0)$ be an $n$-bit NFA. Algorithm 4.9 will jam $FA_0$ by a factor 2 into $2n$-bit NFA $FA_1 = (S_1, \Sigma_1, \delta_1, q_1, F_1)$ iff $FA_0$ is jammable by a factor 2. □
Algorithm 4.9 JAM\((\text{FA}_0, \text{FA}_1, Q, V)\):

Pre: \(n \in \mathbb{Z}^+ \land n \mod 2 = 0 \land \text{JAMMABLE(FA}_0)\)

Post: \(\text{FA}_1\) is a 2-jam of \(\text{FA}_0\)

|| \(Q, V := \emptyset, \emptyset\)
| : \(S_1 := S_0\)
| : \(q_1 := q_0\)
| : \(F_1 := F_0\)
| : \(\Sigma_1 := \mathbb{B}^{2n}\)
| : \(\delta_1 := \emptyset\)
| : \(\text{enqueue}(q_0, Q)\)

\(\text{do } Q \neq \emptyset \rightarrow\)

\(\quad p := \text{dequeue}(Q)\)

\(\quad ; \text{for all } q, a : ((p, a), q) \in \delta_0 \rightarrow\)

\(\quad \quad \text{for all } r, b : ((q, b), r) \in \delta_0 \rightarrow\)

\(\quad \quad \quad S_1 := S_1 \setminus \{q\}\)

\(\quad \quad \quad \delta_1 := \delta_1 \cup \{(p, ab), r\}\)

\(\quad \quad ; \text{as } r \notin V \rightarrow\)

\(\quad \quad \quad V := V \cup \{r\}\)

\(\quad \quad ; \text{enqueue}(r, Q)\)

\(\text{rof}\)

\(\text{rof}\)

\(\text{od}\)

\(\||\)

---

Proof of proposition 4.8.

To show that \(\text{FA}_1\) is a 2-jam of \(\text{FA}_0\) we have to prove that \(\text{FA}_0\) is a 2-stretch of \(\text{FA}_1\) according to definition 2.3.

- First we show that there is a bijection \(\tau : \Sigma_1 \leftrightarrow \Sigma_0^f\), with \(f = 2\). Because \(\Sigma_0^f = (\mathbb{B}^n)^2 = \mathbb{B}^{2n} = \Sigma_1\), there is a trivial bijection between the alphabet of \(\text{FA}_1\) and the \(f\)-closure of the alphabet of \(\text{FA}_0\).

- After the algorithm is executed, \(S_1\) is equal to \(S_0\), \(q_1\) is equal to \(q_0\) and \(F_1\) is equal to \(F_0\), therefore there is an obvious bijection \(\varphi : S_1 \leftrightarrow S_0\). Because \(\text{FA}_0\) is jammable, the states can be divided in set \(T_0\) and the set \(T_1\) according to proposition 4.2.

The algorithm is based on the BFS algorithm and searches for all states \(p\) from set \(T_0\). From state \(p\) it searches for all paths \(\langle (p, a, q), (q, b, r) \rangle\) of length 2 in \(\text{FA}_0\), removes the redundant state \(q\) from \(S_1\) and constructs the transition \((p, ab, r)\) in \(\delta_1\). Therefore, for each transition \((p, ab, r)\) in \(\text{FA}_1\) there is a path of length 2 from \(p\) to \(q\) in \(\text{FA}_0\) with \(a\) and \(b\) as labels so the conditions of definition 2.3 hold.

\(\square\)
Chapter 5

Improvements on the Algorithms

5.0 New Stretch Algorithms

In this section a number of improvements on the original stretch algorithm are presented. The BFS algorithm from the previous chapter is also used in the stretch algorithm below. To stretch NFA $\text{FA}_0$ by a factor $f$ into NFA $\text{FA}_1$, we stretch each newly found transition and insert it into $\text{FA}_1$.

Proposition 5.0. Let $\text{FA}_0 = (S_0, \Sigma_0, \delta_0, q_0, F_0)$ be an $n$-bit NFA. Algorithm 5.1 will stretch $\text{FA}_0$ by a factor $f$ into $\lceil \frac{n}{f} \rceil$-bit NFA $\text{FA}_1 = (S_1, \Sigma_1, \delta_1, q_1, F_1)$.

Algorithm 5.1 BFS-STRETCH($\text{FA}_0, \text{FA}_1, f$):

\begin{verbatim}
| Pre : $n \in \mathbb{Z}^+ \land f \in \mathbb{Z}^+ \setminus \{1\} \land n \mod f = 0$
| Post : $\text{FA}_1$ is an $f$-stretch of $\text{FA}_0$
| $| Q, V ::= \emptyset, \emptyset$
| $| S_1 ::= S_0$
| $| q_1 ::= q_0$
| $| F_1 ::= F_0$
| $| \Sigma_1 ::= \Sigma_0$
| $| \delta_1 ::= \emptyset$
| $| enqueue(q_0, Q)$
| $| do Q \neq \emptyset \rightarrow$
| $| \quad p ::= dequeue(Q)$
| $| \quad do q, a : ((p, a), q) \in \delta_0 \rightarrow$
| $| \quad \ s_{pq}^0 ::= p$
| $| \quad \ I ::= I \cup \{s_{pq}^0, \ldots, s_{pq}^{f-1}\}$
| $| \quad \ s_{pq}^{f} ::= q$
| $| \quad do i ::= 0 to f - 1 \rightarrow \delta_1 ::= \delta_1 \cup \{(s_{pq}^{i}, a[i]), s_{pq}^{i+1}\}$
| $| \quad as q \notin V \rightarrow$
| $| \quad \ V ::= V \cup \{q\}$
| $| \quad enqueue(q, Q)$
| $| \quad sa$
| $| \quad rof$
| $| od$
\end{verbatim}
Proof of proposition 5.0. Our new stretch algorithm is very similar to the stretch algorithm 4.1 of the previous chapter. That algorithm iterates over all transitions in a single for loop. Within this for loop, each transition is stretched. Our new algorithm is based on the BFS algorithm, which discovers each state exactly once. Because we use a for loop to iterate over all transitions of every found state, we also iterate over all transitions in the NFA. Within this for loop we use exactly the same code for the stretching operation. Therefore, algorithm 5.1 will stretch $F_{A_0}$ according to the conditions in definition 2.2.

Because the stretch algorithm can add multiple outgoing transitions with the same label from one state the result of stretching a DFA can be an NFA. We can stretch from a DFA to a DFA if we do not add a transition if a transition with the same label already exists.

Instead of creating it, we follow the transition that’s already present in the stretched DFA. We add the following loop to the code that prevents creation of multiple outgoing transitions with the same label from one state:

\[
i := 0
\]
\[
\textbf{do}
\]
\[
\text{\delta}_1(s, a[i]) \neq \emptyset \rightarrow
\]
\[
s := \text{\delta}_1(s, a[i])
\]
\[
; i := i + 1
\]
\[
\textbf{od}
\]

If transition $(p, a, q)$ is found in the original DFA, a path with label $a[0] \ldots a[f - 1]$ must be added to the stretched DFA. If a part of this path already exists, the loop above travels this path. The loop ends if an outgoing transition with label $a[n]$ does not exist. At that point, $i = n$ and the remaining part of the path is constructed in the DFA.

**Proposition 5.2.** Let $F_{A_0} = (S_0, \Sigma_0, \delta_0, q_0, F_0)$ be an $n$-bit DFA. Algorithm 5.3 will D-stretch $F_{A_0}$ by a factor $f$ into $\frac{n}{f}$-bit DFA $F_{A_1} = (S_1, \Sigma_1, \delta_1, q_1, F_1)$.

**Algorithm 5.3 D-STRETCH($F_{A_0}, F_{A_1}, f$):**

\[\textbf{Pre} : n \in \mathbb{Z}^+ \land f \in \mathbb{Z}^+ \setminus \{1\} \land n \mod f = 0\]
\[\textbf{Post} : F_{A_1} \text{ is an } f\text{-Dstretch of } F_{A_0}\]
\[\left[ Q, V := \emptyset, \emptyset \right.\]
\[; S_1 := S_0\]
\[; q_1 := q_0\]
\[; F_1 := F_0\]
\[; \Sigma_1 := \mathbb{B}^n_f\]
\[; \delta_1 := \emptyset\]
\[; \text{enqueue}(q_0, Q)\]
\[; \textbf{do } Q \neq \emptyset \rightarrow\]
\[\quad p := \text{dequeue}(Q)\]
\[\quad \textbf{for all } q, a : ((p, a), q) \in \delta_0 \rightarrow\]
\[\quad s_0^{pq} := p\]
\[\quad ; i := 0\]

22
Proof. First we have to prove that the result of DStretching a DFA will be a DFA. We can prove this by showing that no outgoing transition from state $s$ with label $a$ is created if there already exists an outgoing transition from $s$ with label $a$, for all $a$ and $s$.

If we find transition $(p, a, q)$ in the original DFA, we have to add a path with label $a[0]a[1] \cdots a[f-1]$ to the stretched DFA. In the algorithm, instead of adding this path immediately, we first travel the first part of this path, if it exists. Note that, because the original automaton is a DFA, there are no two outgoing transitions with the same label from a given state. Therefore, at some point $i$ ($0 \leq i < f$) of the path we are traveling in the stretched DFA, there is no outgoing transition with label $a[i]$. If at that point $i$ we are in intermediate state $i_n$, we can add an outgoing transition with label $a[i]$ from $i_n$ because no transition with that label exists. This proves that the resulting automaton will be a DFA and that the correct transitions are added to the DFA.

But this presents a problem if there is more than one path to the intermediate state $i_n$. Because in that case we would also add more than one path of transitions to the stretched DFA if we add an outgoing transition to state $i_n$. Therefore, we have to prove that there can only be one path to any intermediate state $i_n$.

In the algorithm we only add transitions if a transition with the same label does not exist already. Furthermore, if a new transition $(p, a, q)$ is added, $q$ is a newly introduced state and not an existing state. Therefore, there is only one path to each intermediate state. This concludes the proof.

Note that with a small adjustment, this algorithm can also be used on NFAs to prevent adding more nondeterminism to the NFA by the stretching operation.

During a stretch operation a number of intermediate states is inserted for every transition. The number of states inserted by algorithm 5.3 above does not have to be minimal. See for example figure 5.0, we can see that after a d-stretching operation we can still minimize the DFA.

We can make a small adjustment again to keep the inserted intermediate states minimal. After we have stretched all outgoing transitions from a state $s$ with a factor $f$, we can minimize the number of intermediate states as follows:

If for states $p$ and $q$ at distance $f-1$ from $s$, $\delta(p, a)$ is equal to $\delta(q, a)$ for all $a \in \Sigma$ then either
Figure 5.0: Minimization after a d-stretching operation

$p$ or $q$ can be removed. Of course, if for example $q$ is removed all transitions $((r, a), q) \in \delta$ for all $r \in Q$ have to be changed into $((r, a), p)$.

This process can be repeated for all states at distance $f - 2$ from $s$ to all states at distance 0 from $s$ (s itself). The following algorithm minimizes the number of states at distance $i$ ($0 \leq i < f$) from $s$.

**Proposition 5.4.** Let $FA_0 = (S_0 \cup I, \Sigma_0, \delta_0, q_0, F_0)$ be an f-stretched DFA, with $S_0$ the set of states of the original DFA and $I$ the set of intermediate states. Let $s \in S_0$ and let $D_i$ ($0 \leq i < f$) be the set of states at distance $i$ from $s$. Algorithm 5.5 minimizes the number of states at distance $i$ ($0 \leq i < f$) from $s$.

**Algorithm 5.5 MINIMIZE($FA_0, D_{0..f-1}, s$):**

**Pre:** $D_i$ ($0 \leq i < f$) is the set of states at distance $i$ from $s$

**Post:** The number of states at distance $i$ ($0 \leq i < f$) from $s$ is minimal.

$R$ is the set of states that can be removed.

\[
\begin{align*}
&\text{|| } R := \emptyset \\
&\text{for } i := f - 1 \text{ downto } 1 \rightarrow \\
&\quad \text{for all } p, q : p, q \in D_i \land p \neq q \rightarrow \\
&\quad \quad \text{as } \forall (a : a \in \Sigma_0 : \delta_0(p, a) = \delta_0(q, a)) \rightarrow \\
&\quad \quad \quad R := R \cup \{q\} \\
&\quad \quad \text{for all } r, a : r \in D_{i-1} \land a \in \Sigma_0 : \delta_0(r, a) = q \rightarrow \\
&\quad \quad \quad \delta_0(r, a) = p \\
&\quad \quad \text{rof} \\
&\quad sa \\
&\quad \text{rof} \\
&\quad \text{rof} \\
&\text{||}
\end{align*}
\]

**Proof.** We have to prove that the algorithm minimizes the number of intermediate states introduced by stretching. If for states $p$ and $q$, $\delta(p, a) = \delta(q, a)$ for all symbols $a$, then they essentially serve the same purpose in the DFA and one of them can be removed. This means that if state $q$ is removed, all transitions $(r, a, q)$ have to be changed into $(r, a, p)$, for all states $r$ and symbols $a$. This is why the algorithm moves from all states at distance $f - 1$ to all states at distance 1.

The number of intermediate states cannot be minimized further. We can prove this as follows. If for one symbol $a$, $\delta(p, a) \neq \delta(q, a)$ holds, and we remove either state $p$ or $q$, we also remove a certain path in the DFA or introduce a new path in the DFA, or both.  

\[\square\]
5.1 New Jam Algorithms

In chapter 2 we gave our first definitions of stretching and jamming. Because of the symmetry in these definitions jamming is only defined on the subset of NFAs that are stretched NFAs. In section 4.1, proposition 4.2 we presented the properties that are necessary for an NFA to be jammable by a factor $f$ according to definition 2.3. In the same section we gave an algorithm to detect if an NFA was jammable by a factor 2 and an algorithm to jam an NFA by a factor 2.

In this section we will leave the strict definition of jamming and present an algorithm that can jam any NFA by a factor $f$. We are faced with the problem that some NFAs are not jammable by a factor $f$. See for example figure 5.1. It is easy to see that NFA $FA_0$ is not an $f$-partite NFA (for $f \geq 2$). Therefore, because of proposition 4.2, $FA_0$ is not jammable by a factor $f$.

![Figure 5.1: NFA FA$_0$ = (\{a, b, c\}, \mathbb{B}, \delta_0, a, \{c\})](image)

One solution might be to add ‘dummy’ transitions to make $FA_0$ an $f$-partite NFA. For example, we could transform $FA_0$ into a bipartite NFA with all properties of proposition 4.2 if we add a dummy transition with label $\$ as in figure 5.2(a) or 5.2(b). Of course, $\$ must not be part of the original alphabet.

![Figure 5.2: NFA FA$_{1a}$ and NFA FA$_{1b}$](image)

Next, we can jam the new NFA by a factor 2. NFA $FA_{1b}$, for example, can be jammed by a factor 2 into DFA $FA_2$ of figure 5.3. One problem with this solution is that for every NFA

![Figure 5.3: DFA FA$_2$ = (\{a, b\}, \mathbb{B}^2 \cup \{0\}, \delta_2, a, \{c\})](image)

there is more than one way to add the dummy transitions to transform it into an $f$-partite
NFA. Another problem is that an intermediate NFA is generated. This intermediate NFA might have a very big transition table size.

A slightly different approach to the problem is the following: If we want to jam NFA FA₀ by a factor f into NFA FA₁, we essentially want to make f transitions in one step, instead of one transition at a time. We can achieve this if we find all paths of length f from the start state q₀ to states t₀, . . . , tᵢ. Then we can add all transitions (q₀, wⱼ, tⱼ) to FA₁, where wⱼ is the label of the path from q₀ to tⱼ, (0 ≤ j ≤ i). Next, we repeat the same process for states t₀, . . . , tᵢ, and so on until no more new states are found.

With this solution, problems arise in two situations:

1. If a final state is on a path of length f from state s (excluding the start and end state of the path).

2. If there is a state with no outgoing transitions on a path from state s, and the length of the path is shorter than f.

**Ad 1.** Let there be a path of length n (n < f), in FA₀ from state s to a final state t with label w. A path from s with label w to a final state must be included in the jammed NFA FA₁, otherwise FA₁ will not accept the same set of words as the original NFA FA₀. We can solve this problem by creating a special final state ⊥ in FA₁ with no outgoing transitions. Then we can add transition (t, v$^{f-n}$, ⊥) to FA₁, where $^{f-n}$ is used as ‘padding’ to make the label v$^{f-n}$ exactly size f.

**Ad 2.** Because every state is on a path from start state to end state, a state with no outgoing transitions must be a final state. Therefore, the same solution as in Ad 1. can be used.

To illustrate this approach to jamming an NFA, we give an example.

![Diagram](image)

(a) NFA FA₀ = (\{a, b, c\}, \{0, 1\}, δ₀, a, \{c\})

(b) NFA FA₁ = (\{a, b, c, ⊥\}, \{0, 1, $\}$, δ₁, a, \{c, ⊥\})

Figure 5.4: NFA FA₀ and NFA FA₁, a 2-jam of FA₀

**Example 5.6.** NFA FA₀ of figure 5.4(a) can be jammed by a factor 2 into NFA FA₁ of figure 5.4(b). We do this by finding all paths of length 2 from every state, as described above. For
example, if we look at all paths of length 2 from start state $a$ in FA$_0$ we find a path with label 00 to $a$ itself, we find a path with label 00 to state $b$, and we find a path with label 01 to state $c$. If we look at the paths of length 2 from state $b$ in FA$_0$ we find that on the path with label 11 to state $c$, $c$ is a final state. Therefore, we have to add a transition to FA$_1$ from state $b$ to state $\perp$ with label 1$.$

To find all paths of length $f$ from a give state $p$ we can use the Depth-First Search (DFS) algorithm. Algorithm 5.8 below is a modification of the DFS algorithm found in [CLR01]. In depth-first search, transitions are explored out of the most recent found state. When all transitions out of the most recent state have been explored, the search ‘backtracks’ to earlier found states.

The DFS algorithm is better suited for jamming than the BFS algorithm. If we jam by a factor $f$ we want to find all paths of length $f$ in the NFA. This can be done efficiently by the DFS algorithm because it recursively descends into the NFA and it can stop when it reaches depth $f$.

In the original DFS algorithm a set $V$ is used for states that have already been found. If the search encounters a state that has already been found it does not explore the transitions out of that state. Because we need to find all paths, we also need to explore transitions out of states that have already been found. Therefore, instead of using a set $V$ of found states, we use a variable $c$ to indicate the current depth. If the search reaches depth $f$ we stop searching.

Our algorithm DEPTH-DFS uses a recursive procedure DEPTH-DFS-VISIT to search all paths of length $f$. It is possible to do the search with a non-recursive procedure, but then we would need a stack to store all found states and the paths to these states.

**Proposition 5.7.** Let FA$_0 = (S_0, \Sigma_0, \delta_0, q_0, F_0)$ be an NFA, let $p \in S_0$ and let $d \in \mathbb{N}$. Algorithm 5.8 will traverse FA$_0$ from state $p$ in a depth-first search manner, until depth $d$ is reached.

**Algorithm 5.8 DEPTH-DFS*(FA$_0$, $p$, $d$):**

```plaintext
l := \epsilon
; c := 0
; DEPTH-DFS-VISIT(FA$_0$, $p$, $p$, $d$, $c$, l)
```

**Algorithm 5.9 DEPTH-DFS-VISIT*(FA$_0$, $r$, $p$, $d$, $c$, $l$):**

```plaintext
as $c < d$ →
  for all $q, a : ((p, a), q) \in \delta_0$ →
    DEPTH-DFS-VISIT(FA$_0$, $r$, $q$, $d$, $c + 1$, $la$)
  rof
  sa
```

**Proof of proposition 5.7.** A correctness proof can be found in [CLR01].
Proposition 5.10. Let $FA_0 = (S_0, \Sigma_0, \delta_0, q_0, F_0)$ be an $n$-bit NFA. Algorithms 5.11 and 5.12 will jam $FA_0$ by a factor $f$ into $nf - \text{bit}$ NFA $FA_1 = (S_1, \Sigma_1, \delta_1, q_1, F_1)$.

Algorithm 5.11 JAM($FA_0$, $FA_1$, $f$):

Pre: $n \in \mathbb{Z}^+$
Post: $FA_1$ is an $f$-jam of $FA_0$

|| $Q := \emptyset$
|| $S_1 := \{\bot\}$
|| $q_1 := q_0$
|| $F_1 := F_0 \cup \{\bot\}$
|| $\Sigma_1 := \{x^{n_f - ni} : x \in \mathbb{B}^{ni}, 1 \leq i \leq f\}$
|| $\delta_1 := \emptyset$
|| $l := \epsilon$
|| $c := 0$
|| enqueue$(q_0, Q)$
|| $S_1 := S_1 \cup \{q\}$
|| do $Q \neq \emptyset$
|| $q := \text{enqueue}(Q)$
|| JAM-PATH($FA_0$, $FA_1$, $Q$, $r$, $q$, $d$, $c$, $l$)
|| od
||

Algorithm 5.12 JAM-PATH($FA_0$, $FA_1$, $Q$, $r$, $p$, $d$, $c$, $l$):

|| if $c < d$ →
|| as $p \in F_0$ →
|| $\delta_1 := \delta_1 \cup \{(r, l^{d-c}), \bot\}$
|| sa
|| for all $q, a : (p, a), q \in \delta_0$ →
|| JAM-PATH($FA_0$, $FA_1$, $Q$, $r$, $q$, $d$, $c + 1$, $la$)
|| rof
|| $c = d$ →
|| as $p \notin S_1$ →
|| $S_1 := S_1 \cup \{p\}$
|| enqueue$(p, Q)$
|| sa
|| $\delta_1 := \delta_1 \cup \{(r, l), p\}$
|| fi
||

Proof. The algorithm is based on the DFS algorithm 5.8. The DFS algorithm finds all paths of length $f$ from the start state $q_0$. All end states of the path, or in other words, all states that are at distance $f$ from $q_0$, are added to the queue and for these states the same process is repeated recursively.
The label of each path is recorded in variable $l$, and every time a new transition $(p, a, q)$ is found, the algorithm recursively calls itself with the new label $la$ and new state $q$. This way, the algorithm recursively descends into the NFA.

Variable $c$ indicates the current depth of the path, so if depth $d = f$ is reached, a path of length $f$ from state $r$ to state $p$ with label $l$ has been found. This means a new transition $(r, l, p)$ can be added to the jammed NFA. If a final state is found on a path that does not have length $d = f$, a transition $(r, l, \bot)$ is added to $FA_1$, where $r$ is the first state of the path, $l$ the label, and $\bot$ is the special introduced bottom state.

Therefore, if a string is processed by the jammed NFA, it can travel any path to a final state as it would have done in the original NFA. This means the jammed NFA will accept the same set of strings as the original NFA.

We conclude this proof by proving the correctness of the construction of the alphabet. The original NFA $FA_0$ has an $n$-bit alphabet. The algorithm finds all paths of length $f$, but on each path a final state might be found. Therefore, paths of length $1, 2, \ldots, f$ can be found. Because each symbol in the original NFA is $n$ bits long, a symbol in the new alphabet can have length $n, 2n, \ldots, fn$ bits. The alphabet symbols of the jammed NFA must have bit-length $nf$, therefore padding of $\$ symbols is added to the alphabet symbols if they do not have length $nf$. $\qed$
Chapter 6

Implementation

In chapter 3 we presented theoretical results for our stretching and jamming operations. These theoretical results show that by stretching a DFA or NFA we can potentially reduce the amount of memory needed for the transition table. But because during a stretching operation every transition is stretched into two or more sequential transitions, we expect that the time to recognize a string will be longer after the stretching operation. Jamming an automaton on the other hand will, in most cases, increase the amount of memory needed for the transition table, but we expect that it will reduce the average string recognition time.

To be able to verify our assumptions we have implemented our stretch and jam algorithms. In this chapter we will present the details of that implementation. The implementation of the algorithms is used to obtain practical results by performing benchmarking tests. The results of the benchmarking tests are presented in the next chapter.

6.0 Overview

In this section we present the general design decisions we took for the implementation of our algorithms. We have decided to use C++ as implementation language. The reasons for this are that C++ is a flexible and efficient general purpose programming language that supports Object-Oriented design. The combination of C++ with the Standard Template Library (STL) makes it possible to create elegant and efficient implementations.

Our design is centered around seven classes, presented in the next section. Two of these classes, namely the DFA and NFA classes, have methods for stretching and jamming operations. This enables us to create a DFA or NFA object and stretch or jam it by a certain factor.

Note that we only implement the algorithms presented in chapter 5, and not the less efficient algorithms in chapter 4. Implementing less efficient algorithms will not lead to more reliable benchmarking results so implementation of those algorithms is not needed.

6.1 Classes

In this section we present seven classes that together implement a system for stretching and jamming DFAs and NFAs. In figure 6.0, a UML class diagram is presented that shows the relationship between our classes. For an explanation of UML notation, see [Fow04].
Because we want to perform operations on DFAs and NFAs we have implemented the classes DFA and NFA. In our definition the only difference between DFAs and NFAs is in the transition function, which is implemented by a transition table in our design. In a DFA the transition function is defined as a partial function \( \delta : Q \times \Sigma \rightarrow Q \), with \( Q \) the set of states and \( \Sigma \) the set of alphabet symbols. In the transition table of a DFA there is a row for each state, and a column for each alphabet symbol. This way, there is a cell for each combination of a state and an alphabet symbol. Because the range of the transition function is the set of states, each cell of the transition table contains a state, or it is empty if the function is not defined for that row and column.

In an NFA the transition function is defined as a partial function \( \delta : Q \times \Sigma \rightarrow \mathcal{P}(Q) \). Therefore, in the transition table of an NFA there is also a row for each state and a column for each alphabet symbol. The difference is in the cells of the transition table. Because the range of the transition function is \( \mathcal{P}(Q) \), the powerset of states, each cell contains a set of states. This set is empty if the transition function is not defined for the particular row and column.

![Class diagram]

Figure 6.0: Class diagram

Because of the similarities between DFAs and NFAs our classes DFA and NFA are derived from an abstract parent class FA<ValType>. The abstract parent class FA<ValType> has a template parameter ValType for the type of the cell of the transition table. This parameter is shown in the dashed box in the top-right corner of the class. The type of ValType is typename, indicating that the argument to the parameter can be a user-defined or a C++ primitive type. We will sometimes omit the template parameter if we talk about FA<ValType>, and just write FA.

As we saw earlier, each cell in the transition table of a DFA contains a state. Therefore the class DFA is a subclass of the class FA<ValType>, where the type State is used as argument for the class parameter. In the class diagram this is denoted as FA<ValType::State>. Methods, attributes and other details of the class DFA are given in section 6.1.4.

Each cell in the transition table of an NFA contains a set of states, so in the case of our
class NFA we use set<State> as argument for the parent class FA<ValType>. In section 6.1.4 the details of class NFA are discussed.

We also have a class JammedDFA that is, like DFA, a subclass of FA<ValType::State>. The reasons for creating a separate class for jammed DFAs are given in section 6.1.6.

In our class diagram, it is shown that our classes DFA and NFA are subclasses of respectively FA<ValType::State> and FA<ValType::set<State>>. These classes, in turn, are a derivation of the class FA. The keyword ≪bind≫ denotes that these classes are bound to class FA through template parameter instantiation with the correct type. Class FA<ValType> is treated in section 6.1.3.

Every FA has a transition table, therefore the class FA<ValType> has an association with the class TransitionTable<ValType>. The template parameter ValType of the transition table class is instantiated with the template parameter ValType of FA<ValType>. This way, the class DFA instantiates FA with the argument State, and on its turn this class passes on the type State to instantiate TransitionTable. As we will see, TransitionTable and its parent class will also pass on this variable.

Our DFA and NFA classes both have one transition table, but our JammedDFA class has multiple transition tables depending on the jam factor $f$. This explains the multiplicity of the association between FA and TransitionTable. TransitionTable will be discussed in section 6.1.2.

A transition table can be implemented by a matrix, with a row for each state and a column for each alphabet symbol. We have chosen to create a Matrix<RowType, ColType, ValType> class, which has template parameters for the type of rows, the type of columns and the type of cells. Class TransitionTable gets the parameter for the type of cells from FA and passes it on to Matrix<RowType, ColType, ValType>. Furthermore, TransitionTable instantiates parameter RowType with the type State and ColType with the type AlphabetSize. See section 6.1.1 for more details about the Matrix class.

A matrix is a 2-dimensional vector, so our last class is a vector class Vec<ColType, ValType>. Vec has template parameter VecType for the type of rows, and ValType for the type of cells. Matrix instantiates VecType with the type RowType. Because we need a 2-dimensional vector, parameter ValType is instantiated with a vector itself, namely Vec<ColType, ValType>. In the next section we start the individual discussion of our classes with class Vec.

6.1.0 Vec

Because we want to use vectors we have chosen to create a vector class, called Vec<ColType, ValType>. As explained before, it has template parameter VecType for the type of rows, and ValType for the type of cells. Fortunately, the STL comes with its own vector class but one problem with this class is that it does not provide range checking. This means that if a subscript argument is out of range, the vector will not give an error but return a random value.

<table>
<thead>
<tr>
<th>Class Vec&lt;ColType, ValType&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;typename VecType, typename ValType&gt;</td>
</tr>
<tr>
<td>class Vec : public vector&lt;ValType&gt;</td>
</tr>
<tr>
<td>+ Vec() Default constructor</td>
</tr>
<tr>
<td>+ Vec(VecType elem) Constructor with number of elements</td>
</tr>
</tbody>
</table>
Therefore we based our class Vec on the class Vec from [Str00]. This class throws an exception if the subscript argument is out of range. By using exceptions we are certain that either the correct value is returned or we are notified that something is wrong.

Besides range checking we added other functionality such as a virtual Print method which makes it possible to print the vector to standard output. Table 6.1 lists all the methods of Vec<ColType, ValType>. The first row of the table lists the name of the class. The second row contains the definition of the class. After that, a list of methods and attributes follow. The first column of each row shows the definition of the method or attribute and the second column contains a small comment to explain its functionality.

### 6.1.1 Matrix

The class Matrix<RowType, ColType, ValType> is a 2-dimensional vector, therefore it is defined as a subclass of Vec<RowType, Vec<ColType, ValType>>. This means that Matrix is derived from Vec and it uses RowType as argument for the rows, and for the columns another vector Vec which takes ColType for the rows and ValType for the values of the cells. Matrix overloads the virtual Print function so matrices can also be printed to standard output.

The table below shows all methods of the class Matrix. It’s very similar to Vec, with a few additional methods that speak for themselves.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Matrix()</td>
<td>Default constructor</td>
</tr>
<tr>
<td>+ Matrix(RowType rows, ColType columns)</td>
<td>Constructor with number of rows and columns</td>
</tr>
<tr>
<td>+ Matrix(RowType rows, ColType columns,</td>
<td>Constructor with default value</td>
</tr>
<tr>
<td>const ValType &amp;value)</td>
<td></td>
</tr>
<tr>
<td>+ virtual ~Matrix()</td>
<td>Destructor</td>
</tr>
<tr>
<td>+ RowType NumRows() const</td>
<td>The number of rows</td>
</tr>
<tr>
<td>+ ColType NumColumns() const</td>
<td>The number of columns</td>
</tr>
<tr>
<td>+ void Resize(RowType rows)</td>
<td>Change the number of rows</td>
</tr>
<tr>
<td>+ void Resize(RowType rows, ColType columns)</td>
<td>Change number of rows and columns</td>
</tr>
<tr>
<td>+ void Resize(RowType rows, ColType columns,</td>
<td>Change number of rows</td>
</tr>
<tr>
<td>const ValType &amp;value)</td>
<td></td>
</tr>
</tbody>
</table>
6.1.2 TransitionTable

The class TransitionTable<ValType> is used as implementation of the transition functions for both DFAs and NFAs. A transition function of a DFA as well as an NFA take a state and alphabet symbol as argument. Therefore, TransitionTable<ValType> is derived from Matrix<State, AlphabetSize, ValType>. Type State is used as the type of rows and AlphabetSize is used for the columns.

Template parameter ValType is passed on to class Matrix for the type of cells and functions as the return type of the transition function. This way, we can have different return types for transition functions of DFAs and NFAs.

Of course, TransitionTable has constructor and destructor methods. Note that the parameter AlphabetSize for the constructors is the alphabet size in bits. So, TransitionTable(10,2) will result in a transition table with 10 states and a 2 bits alphabet, which makes the number of columns equal to $2^2 = 4$.

Besides some constructor and destructor methods we also have a NextState function. This function returns the value of the correct cell for a state and alphabet symbol. We also have AddTransition to add transitions and several Resize functions.

To get or set the number of states or the alphabet size, the methods NumStates and AlphabetBitSize are available. Note that the argument to AlphabetBitSize is also the alphabet size in bits. Method AlphabetDecSize() can be used to retrieve the actual number of columns in the transition table.

The total number of transitions in the transition table, or in other words the total number of cells that are not empty, can be retrieved by NumTransitions(). The size of a transition table is defined as the number of states multiplied by the number of alphabet symbols, therefore Size() returns NumStates() * AlphabetDecSize(). We sometimes want to know how much the transition table is filled with transitions. The transition density is defined as the number of transitions of the transition table divided by the total number of possible transitions. Density() returns the transition density in percent.

The class TransitionTable also has a few private attributes to store information. For example, the number of transitions can be retrieved without checking each cell of the transition table.
AlphabetSize alphabetSize, const ValType &value)
+ ~TransitionTable() Destructor
+ ValType NextState(State inputState, AlphabetSize inputSymbol) Transition function
+ int AddTransition(State inputState, AlphabetSize inputSymbol, const ValType &nextState) Add a transition to the transition table
+ void Resize(State numStates) Change the number of states
+ void Resize(State numStates, AlphabetSize alphabetSize) Change the number of states and alphabet size
+ void Resize(State numStates, AlphabetSize alphabetSize, const ValType &value) Change the number of states, alphabet size with default value for added cells
+ void NumStates(State numStates) Set the number of states
+ void AlphabetBitSize(AlphabetSize alphabetSize) Set the alphabet size (in bits)
+ AlphabetSize AlphabetBitSize() const Alphabet size (in bits)
+ AlphabetSize AlphabetDecSize() const Alphabet size (decimal value)
+ State NumStates() const Number of states
+ unsigned long NumTransitions() Number of transitions
+ unsigned long Size() const Transition table size
+ unsigned long Density() const Transition table density
- AlphabetSize alphabetBitSize_ Alphabet size (in bits)
- AlphabetSize alphabetDecSize_ Alphabet size (decimal value)
- State numStates_ Number of states
- unsigned long numTransitions_ Number of transitions

Table 6.3: Class TransitionTable

6.1.3 FA

FA<ValType> is an abstract parent class for our DFA, JammedDFA and NFA classes. This means that FA itself will not be instantiated but only its subclasses. FA accommodates all shared functionality between DFAs and NFAs. Table 6.4 lists all methods and attributes of FA.

<table>
<thead>
<tr>
<th>Class FA&lt;ValType&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;typename ValType&gt;</td>
</tr>
<tr>
<td>class FA</td>
</tr>
<tr>
<td>+ FA() Default constructor</td>
</tr>
<tr>
<td>+ FA(State numStates, AlphabetSize alphabetSize = BYTE_SIZE) Constructor with number of states and alphabet size</td>
</tr>
<tr>
<td>+ FA(State numStates, AlphabetSize alphabetSize, State startState, set&lt;State&gt; finalStates) Constructor with start state and final states</td>
</tr>
<tr>
<td>+ virtual ~FA() Destructor</td>
</tr>
<tr>
<td>+ virtual void AddTransition(State inState, AlphabetSize inputSymbol, State outState)= 0 Add a transition</td>
</tr>
</tbody>
</table>
Both DFAs and NFAs are defined as a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where \(Q\) is a finite set of states, \(\Sigma\) is the alphabet, \(q_0\) is the start state and \(F\) is a subset of \(Q\) whose elements are final states. The set of states \(Q\) and the alphabet \(\Sigma\) aren’t implemented with sets, but are defined automatically by the dimensions of the transition table. In case of a transition table of \(m\) rows and \(n\) columns, we automatically have defined states numbered \([0..m]\) and alphabet symbols.
[0..n]. The start state is implemented by a private attribute `startState_` and can be retrieved and modified by public methods. Similarly, the set of final states is implemented by a private set of states.

As said in section 6.1, in our definition the difference between DFAs and NFAs is in their transition functions. Both can be implemented by our transition table class though, therefore `FA<ValType>` has an association with `TransitionTable<ValType>`. This association is implemented by a pointer to a transition table object. A transition table object is created in the constructor and can be modified through public methods. The constructors of `FA<ValType>` pass `ValType` to instantiate the transition table with the correct type.

Besides constructors and a destructor, `FA` has public methods to add transitions, to resize the transition table, and to get and set the number of states and alphabet size. Note that some of these methods can only be implemented by subclasses therefore they are pure virtual methods.

Furthermore, `FA` has a private attribute `currentState_` which is initialized with the start state by the constructor and is modified by the method `NextState`. This way, function `NextState` can be called with only an alphabet symbol as argument, `currentState_` is used as input state for the transition and updated accordingly.

To facilitate stretching and jamming, `FA` has some private attributes to indicate if it is stretched or jammed, and by which factor.

### 6.1.4 DFA

Class DFA is a subclass of the abstract class `FA<State>`. `FA<State>` is instantiated with template argument `State` because cells in a DFA are of type `State`. See table 6.5 for the public methods of DFA.

<table>
<thead>
<tr>
<th>Class DFA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>class</strong> DFA : public FA&lt;State&gt;</td>
</tr>
<tr>
<td>+ DFA()</td>
</tr>
<tr>
<td>+ DFA(State numStates, AlphabetSize alphabetSize)</td>
</tr>
<tr>
<td>+ DFA(State numStates, AlphabetSize alphabetSize, State startState, set&lt;State&gt; finalStates)</td>
</tr>
<tr>
<td>+ ~DFA()</td>
</tr>
<tr>
<td>+ virtual State NextState(AlphabetSize inputSymbol)</td>
</tr>
<tr>
<td>+ virtual void Resize(State numStates)</td>
</tr>
<tr>
<td>+ virtual void Resize(AlphabetSize alphabetSize)</td>
</tr>
<tr>
<td>+ virtual void AddTransition(State inState, AlphabetSize inputSymbol, State outState)</td>
</tr>
<tr>
<td>+ DFA *DStretch(StretchFactor stretchFactor, bool reverse=false) const</td>
</tr>
<tr>
<td>+ void DStretchState(DFA *stretchedDFA, queue&lt;State&gt; &amp;Q, vector&lt;State&gt; &amp;V, State startState, StretchFactor stretchFactor, bool reverse=false) const</td>
</tr>
<tr>
<td>+ DFA *MinDStretch(StretchFactor stretchFactor, bool reverse=false) const</td>
</tr>
<tr>
<td>+ void MinDStretchState(DFA *stretchedDFA, ...)</td>
</tr>
</tbody>
</table>
queue<State> &Q, vector<State> &V, State startState, StretchFactor stretchFactor, bool reverse=false) const

+ void MinimizeStretch(const vector<set<State>> &intermediateStates, StretchFactor stretchFactor) Minimize a stretching operation

+ void DeleteStates(const set<State> &deleteStates) Delete states

+ JammedDFA *Jam(StretchFactor jamFactor, vector<State> &newState, bool reverse=false) const

Jam the transition table

+ void JamPath(JammedDFA *MyJammedDFA, queue<State> &Q, vector<State> &V, State rootState, State currentState, StretchFactor currentDepth, StretchFactor jamFactor, AlphabetSize currentSymbol, bool reverse=false) const

Jam paths in the transition table

+ Vec<State, Vec<State, AlphabetSize>> FindPath(State len, State startState) const

Find all paths of a certain length

+ void FindPath(State len, State currentState, State currentDepth, Vec<State, Vec<State, StretchFactor>> &vecResult, Vec<State, AlphabetSize> vecPath) const

Find all paths of a certain length

+ Vec<State, AlphabetSize> RandomPath(State len, State startState, long *idum, bool final=false) const

Find a random path

+ void RandomPath(State len, State currentState, State currentDepth, bool &stop, Vec<State, AlphabetSize> &vecResult, Vec<State, AlphabetSize> vecPath, long *idum, bool final=false) const

Find a random path

+ unsigned long TransitionTableByteSize() const

Number of bytes used for the transition table

+ ostream& Print(ostream& os=cout) const

Print the transition table

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue&lt;State&gt; &amp;Q, vector&lt;State&gt; &amp;V, State startState, StretchFactor stretchFactor, bool reverse=false) const</td>
<td>from a given state</td>
</tr>
<tr>
<td>+ void MinimizeStretch(const vector&lt;set&lt;State&gt;&gt; &amp;intermediateStates, StretchFactor stretchFactor)</td>
<td>Minimize a stretching operation</td>
</tr>
<tr>
<td>+ void DeleteStates(const set&lt;State&gt; &amp;deleteStates)</td>
<td>Delete states</td>
</tr>
<tr>
<td>+ JammedDFA *Jam(StretchFactor jamFactor, vector&lt;State&gt; &amp;newState, bool reverse=false) const</td>
<td>Jam the transition table</td>
</tr>
<tr>
<td>+ void JamPath(JammedDFA *MyJammedDFA, queue&lt;State&gt; &amp;Q, vector&lt;State&gt; &amp;V, State rootState, State currentState, StretchFactor currentDepth, StretchFactor jamFactor, AlphabetSize currentSymbol, bool reverse=false) const</td>
<td>Jam paths in the transition table</td>
</tr>
<tr>
<td>+ Vec&lt;State, Vec&lt;State, AlphabetSize&gt;&gt; FindPath(State len, State startState) const</td>
<td>Find all paths of a certain length</td>
</tr>
<tr>
<td>+ void FindPath(State len, State currentState, State currentDepth, Vec&lt;State, Vec&lt;State, StretchFactor&gt;&gt; &amp;vecResult, Vec&lt;State, AlphabetSize&gt; vecPath) const</td>
<td>Find all paths of a certain length</td>
</tr>
<tr>
<td>+ Vec&lt;State, AlphabetSize&gt; RandomPath(State len, State startState, long *idum, bool final=false) const</td>
<td>Find a random path</td>
</tr>
<tr>
<td>+ void RandomPath(State len, State currentState, State currentDepth, bool &amp;stop, Vec&lt;State, AlphabetSize&gt; &amp;vecResult, Vec&lt;State, AlphabetSize&gt; vecPath, long *idum, bool final=false) const</td>
<td>Find a random path</td>
</tr>
<tr>
<td>+ unsigned long TransitionTableByteSize() const</td>
<td>Number of bytes used for the transition table</td>
</tr>
<tr>
<td>+ ostream&amp; Print(ostream&amp; os=cout) const</td>
<td>Print the transition table</td>
</tr>
</tbody>
</table>

Table 6.5: Class DFA

DFA has some constructors and a destructor and it implements the pure virtual methods of class FA. Furthermore, it has some methods that implement the operations this thesis is all about: the stretching and jamming operations.

The implementation of algorithm 5.3 is divided into several public methods. Method MinDStretch(StretchFactor stretchFactor, bool reverse) implements algorithm 5.3 and it also performs the minimization step as proposed in algorithm 5.5. MinDStretch can be called on a DFA object and it returns a pointer to a new DFA object, which is a stretch of the original DFA. Parameter StretchFactor is used to indicate the stretch factor. Parameter reverse is used to indicate if the alphabet symbol of each transition must be reversed before stretching. By default alphabet symbols are not reversed and therefore the low order bits are used first in the stretching operation. If reverse is true, the high order bits are used first in stretching.

The source code of MinDStretch is listed in Appendix A. MinDStretch uses several other methods that perform different tasks. MinDStretch calls method MinDStretchState for every state. This method stretches all transitions from a given state.

Method MinimizeStretch is an implementation of algorithm 5.5 and it is called by MinD-
StretchState to minimize the number of introduced states. To delete the states that are not needed anymore after minimization, method DeleteStates is used. The source code for all these methods is listed in Appendix A.

Algorithms 5.11 and 5.12 are implemented by methods Jam and JamPath, respectively. MinDStretch returns a pointer to an object of type JammedDFA. See section 6.1.6 for a discussion of the JammedDFA class.

Like MinDStretch, Jam(StretchFactor jamFactor, vector<State> &newState, bool reverse) can be called with the stretch factor and a boolean reverse. To indicate which state in the original DFA corresponds to which state in the jammed DFA, method Jam writes to a vector newState. If newState[n] is not defined for state n, this state has been removed by the jamming operation. If newState[n]=m, state n in the original DFA corresponds to state m in the jammed DFA. Appendix A also contains the source code of these methods.

To facilitate the benchmarking process of the next chapter, DFA has some methods to find different paths of transitions in the DFA.

### 6.1.5 NFA

Our NFA class is very similar to the DFA class of the previous section. NFA is a subclass of the abstract class FA<set<State>>. NFA instantiates FA with template argument set<State>, because every cell in the transition table contains a set of states.

Because we will only perform benchmarking tests on DFAs, we have not implemented all algorithms of chapter 5 for NFAs. Method DStretch implements algorithm 5.3, the source code of this method is listed in Appendix A. An overview of the methods of NFA is shown in table 6.6.

<table>
<thead>
<tr>
<th>Class NFA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>class</strong> NFA : public FA&lt;set&lt;State&gt;&gt;</td>
</tr>
<tr>
<td>+ NFA()</td>
</tr>
<tr>
<td>+ NFA(State numStates, AlphabetSize alphabetSize)</td>
</tr>
<tr>
<td>+ NFA(State numStates, AlphabetSize alphabetSize, State startState, set&lt;State&gt; finalStates)</td>
</tr>
<tr>
<td>+ ~NFA()</td>
</tr>
<tr>
<td>+ virtual set&lt;State&gt; NextState( const AlphabetSize inputSymbol)</td>
</tr>
<tr>
<td>+ virtual void Resize(State numStates)</td>
</tr>
<tr>
<td>+ void Resize(AlphabetSize alphabetSize)</td>
</tr>
<tr>
<td>+ virtual void AddTransition(State inState, AlphabetSize inputSymbol, State outState)</td>
</tr>
<tr>
<td>+ NFA ∗DStretch(StretchFactor stretchFactor, bool reverse=false) const</td>
</tr>
<tr>
<td>+ void AddStates(State numStates=1)</td>
</tr>
</tbody>
</table>

Table 6.6: Class NFA
6.1.6 JammedDFA

In section 5.1 we introduced a new way of jamming. In that new approach, if the jam factor is $f$, we find all paths of length $f$ and add those paths to the jammed NFA in a recursive manner. A problem arises if we find a final state on a path, because then we need to add the path to the jammed NFA while it might not have length $f$. Therefore, not all alphabet symbols have length $f$, and it cannot be represented anymore by a simple alphabet in which all symbols have the same length.

In the same section we proposed a solution where we added ‘padding’ to make the label of the path length $f$. For the implementation of this jam algorithm, we could choose the same approach. But this means that if we want to use an 8-bit alphabet, we cannot use 8-bit C-style characters as input symbol because we would need to reserve some bits for the padding. Furthermore, every time a symbol is used as input we have to check if it contains padding, and possibly perform some calculations before we can choose the right column in the transition table. Therefore, a direct implementation of the padding will increase both memory usage and string recognition time.

A different approach is the following: See the example of figures 5.4(a) and 5.4(b), in chapter 5. During the jamming operation we need to add a transition with label 1$ from state $b$ to state $\bot$ to $FA_1$. Preferably, we would like to forget about the padding and just add a transition $(b, 1, \bot)$ to $FA_1$.

The decimal value of 1-bit symbol 1 is 1 and therefore it corresponds to column 1 in the transition table. But in the 2-bit transition table of $FA_1$, column 1 is already used by symbol 01, the 2-bit symbol which decimal value equals 1. This means that we cannot use the 2-bit transition table of $FA_1$ to add transition $(b, 1, \bot)$ or we would introduce ambiguity. The only other solution then is to create a separate transition table for all 1-bit symbols. This way, a transition $(p, 01, q)$ can added to the normal 2-bit transition table, and a transition $(r, 1, s)$ can added to the extra 1-bit transition table.

We can implement this solution with a vector of transition tables. An $n$-bit NFA that is jammed by a factor $f$ will result in an NFA with a vector of size $f - 1$. This vector contains 1 normal transition table and $f - 2$ extra transition tables. Each vector element $i$ contains a $((i + 1) \ast n)$-bit transition table, for $0 \leq i < f$.

The advantage of this solution is that no padding needs to be added. To choose the right transition table though, we have to indicate how many bits the symbol uses. Alternatively, it can be left up to the user of the class to indicate which transition table it wants to use.

Another optimization can be made for this solution. We saw before that we needed to add transition $(b, 1, \bot)$ to the jammed NFA. In fact, all transitions that need to be added to the extra transition tables will end in the special state $\bot$. This means that we do not need to use type State for the cells of the extra transition tables, but we can use just a single bit for each cell. The conclusion is that this solution will minimize memory usage and string recognition time in comparison with the first solution.

This is why we introduce a special class JammedDFA that contains a vector of transition tables. We have not created a similar class for NFAs because the benchmarking tests of the next chapter will only involve DFAs. The class JammedDFA is very similar to our DFA class of section 6.1.4. Therefore we only discuss the methods that are unique to the class JammedDFA here.

Instead of a pointer to a TransitionTable object, our JammedDFA class has a pointer to a vector of TransitionTable objects. Memory for these transition tables is reserved during
construction time.

Furthermore, JammedDFA has two methods to retrieve the alphabet size of all transition tables together, TotalAlphabetBitSize() and TotalAlphabetDecSize() respectively. In short, all methods that are prefixed with the word Total refer to methods that use all transition tables. Methods that are not prefixed with Total only refer to the normal transition table.

The table below shows all public methods and the protected attribute jammedTransitionTable, which is a pointer to a vector of transition tables.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ JammedDFA()</td>
<td>Default constructor</td>
</tr>
<tr>
<td>+ JammedDFA(State numStates, AlphabetSize alphabetSize, StretchFactor jamFactor)</td>
<td>Constructor with number of states and alphabet size</td>
</tr>
<tr>
<td>+ JammedDFA(State numStates, AlphabetSize alphabetSize, StretchFactor jamFactor, State startState, set&lt;State&gt; finalStates)</td>
<td>Constructor with start state and final states</td>
</tr>
<tr>
<td>+ ~JammedDFA()</td>
<td>Destructor</td>
</tr>
<tr>
<td>+ virtual State NextState(AlphabetSize inputSymbol)</td>
<td>Next state function</td>
</tr>
<tr>
<td>+ State NextState(AlphabetSize inputSymbol, StretchFactor symbolSize)</td>
<td>Next state function</td>
</tr>
<tr>
<td>+ virtual void Resize(State numStates)</td>
<td>Resize the transition table</td>
</tr>
<tr>
<td>+ void Resize(State numStates, AlphabetSize index)</td>
<td>Resize the transition table</td>
</tr>
<tr>
<td>+ virtual void Resize(AlphabetSize alphabetSize)</td>
<td>Resize the transition table</td>
</tr>
<tr>
<td>+ virtual void AddTransition(State inState, AlphabetSize inputSymbol, State outState)</td>
<td>Add a transition</td>
</tr>
<tr>
<td>+ void AddTransition(State inState, AlphabetSize inputSymbol, AlphabetSize symbolSize, State outState)</td>
<td>Add a transition</td>
</tr>
<tr>
<td>+ AlphabetSize TotalAlphabetBitSize()const</td>
<td>The total number of bits for the alphabet</td>
</tr>
<tr>
<td>+ AlphabetSize TotalAlphabetDecSize()const</td>
<td>The total number of alphabet symbols</td>
</tr>
<tr>
<td>+ inline State TotalNumStates()const</td>
<td>The total number of states</td>
</tr>
<tr>
<td>+ unsigned long TotalNumTransitions()const</td>
<td>The total number of transitions</td>
</tr>
<tr>
<td>+ unsigned long TotalTransitionTableDensity()const</td>
<td>The total transition table density</td>
</tr>
<tr>
<td>+ unsigned long TotalTransitionTableByteSize()const</td>
<td>The total number of bytes of the transition table</td>
</tr>
<tr>
<td>+ inline State NumStates()const</td>
<td>Return the number of states</td>
</tr>
<tr>
<td>+ inline AlphabetSize AlphabetBitSize()const</td>
<td>Return the alphabet size (in bits)</td>
</tr>
<tr>
<td>+ inline AlphabetSize AlphabetDecSize()const</td>
<td>Return the alphabet size (decimal value)</td>
</tr>
<tr>
<td>+ inline unsigned long NumTransitions()const</td>
<td>Return the number of transitions</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td><code>inline unsigned long TransitionTableSize() const</code></td>
<td>Return the size of the transition table</td>
</tr>
<tr>
<td><code>inline unsigned long TransitionTableDensity() const</code></td>
<td>Return the transition table density</td>
</tr>
<tr>
<td><code>inline TransitionTable&lt;State&gt;&amp; operator[](const StretchFactor index)</code></td>
<td>Return a transition table</td>
</tr>
<tr>
<td><code>ostream&amp; Print(ostream&amp; os=cout) const</code></td>
<td>Print the transition table of a JammedDFA</td>
</tr>
<tr>
<td>#Vec&lt;StretchFactor, TransitionTable&lt;State&gt;&gt; *jammedTransitionTable_</td>
<td>Vector of transition tables</td>
</tr>
</tbody>
</table>

Table 6.7: Class JammedDFA
Chapter 7

Benchmarking

7.0 Overview

Surprisingly, there is very little literature available on the subject of performance evaluation. [Jai91] lists a number of steps that are common to all performance evaluation projects. We have based our project on these steps to be able to take a systematic approach to the performance evaluation of the stretching and jamming operations. Below we have listed a number of steps and the proposed realization for our evaluation.

1. System under Study and Goals
   The system under study is finite state automata. More precisely, performance benchmarks are carried out on DFAs, and stretched and jammed DFAs. We have chosen to exclude NFAs for now and focus on DFAs which enables us to look at DFAs in more detail. We believe that performance benchmarks on NFAs will give similar results but this is left as future work. The goal of the benchmarking study is to measure the effect of stretching on the performance of a DFA. The same effect will be measured for jamming. Specifically, we want to know in which cases stretching and jamming have a positive effect on the performance.

2. Task of the system
   The task of a DFA is to recognize strings. To do this, the DFA uses a transition function that takes a state and a symbol of the string and returns a state. The first transition starts in the start state, takes the first symbol of the string as input and results in the next state. This next state is used in combination with the next symbol of the string to go to another state. This way, a string can be recognized by using consecutive symbols of the string as input for the transition function. A string is accepted if all symbols of the string are used and the last resulting state is a final state. Otherwise, the string is rejected. Of course, a stretched or jammed DFA should accept the same set of strings as the original DFA.

3. Selected Metrics
   To implement the transition function of a DFA we use a transition table representation. We want to measure the memory usage of the transition table of a DFA. This way, we can compare the memory usage before and after stretching or jamming. Less memory
usage means better performance. We also want to measure the amount of time needed to recognize a string. This allows us to compare the string recognition time before and after stretching or jamming. Less time needed to recognize a string means better performance.

4. Parameters
The parameters below affect the benchmarking study. They are divided into system parameters and workload parameters. System parameters are fixed, so they do not change between different measurements. The workload parameters are varied to measure the performance under different conditions. We do not change all workload parameters for both memory usage measurements and string recognition time measurements, therefore they are divided into three categories.

**System parameters:**
- Hardware: Linux/x86 and Mac OS X/PowerPC
- Programming Language: C++/STL
- Implementation details

**General workload parameters:**
- Stretch factor
- Jam factor

**Workload parameters for memory usage measurements:**
- Alphabet size
- Number of states
- Transition table density

**Workload parameters for string recognition time measurements:**
- Alphabet size
- Input string length

5. Factors to study
All workload parameters can be varied for both experiments, but some will have more impact on the performance than others. The general workload parameters are likely to affect performance the most, therefore they can be varied in both measurements. For memory usage measurements the transition table density, alphabet size and number of states (in that order) will influence the performance the most. The number of states of the DFA or the transition density will probably not affect the string recognition time measurements. Therefore, we use alphabet size and input string length as parameters in those experiments. Below, we list the different levels we take for the different parameters. We expect the range to be broad enough to be able to give reliable results.

**General workload parameters:**
- Stretch factor: 2, 4, 8
- Jam factor: 2, 4, 8
Workload parameters for memory usage measurements:
- Alphabet size: \(2^1, 2^2, 2^4, 2^8\)
- Number of states: 10, 100, 1000
- Transition table density: 1% – 100%

Workload parameters for string recognition time measurements:
- Alphabet Size: \(2^1, 2^2, 2^4, 2^8\)
- Input string length: 8 bytes, 16 bytes, 32 bytes

6. Evaluation Technique
We have an implementation in C++ to construct DFAs and we have implemented the stretch and jam algorithms, so we can perform measurements with these implementations. Evaluation of memory usage will be done by measuring the memory usage of the transition table. This way, we can compare the memory usage for different DFAs before and after stretching and jamming. To evaluate the string recognition time, we will use strings of different length as input for the DFAs. This enables us to compare the time it takes for a DFA to recognize a string before and after stretching and jamming.

7. Workload
The measurements will be carried out under different workloads that are representative for real life situations. The workload for memory usage measurements will consist of random generated DFAs while varying the factors we have chosen. For the measurement of string recognition time we will generate input strings by constructing random paths of transitions of varying length in random generated DFAs.

8. Design of experiments
First we will generate a number of random DFAs with a specific alphabet size, number of states and transition table density. Next, we will stretch and jam the generated DFA by different factors and compare the memory usage for the transition tables with the original DFAs. We can repeat this experiment for DFAs with different alphabet size, number of states and transition table density.

After that, we will generate a number of random paths in a random DFA. We will measure the time it takes for the stretched and jammed DFA to process the sequential transitions of the path. We can compare these times to the time it takes for the original DFA to process the transitions. Specifically, we will measure the time between the input of the first symbol and the return of the last resulting state. The resulting state before and after stretching and jamming will be compared because they need to be the same.

9. Analyzation and interpretation of data
To compare the memory usage of a random DFA and its stretched or jammed version, we take the arithmetic mean of the memory usage of the transition table over a number of experiments. If, for example, for one experiment the mean memory usage of a stretched DFA is less than the mean memory usage of the original DFA, the stretched DFA has a better performance. Of course, we have to take into account the variance in the results, so for every memory usage measurement we will also keep track of the lowest and highest memory usage. This way we can discover the effect of stretching and jamming on the memory usage for DFAs of different alphabet size, number of states and density.

To analyze string recognition time we can place timers so we can measure the time between the input of a string and the recognition of that string. To compare the string
recognition time of a string of specific length for a DFA and its stretched or jammed counterpart, we can take the arithmetic mean over a number of experiments. Also this time we need to take into account to variance in the results. This way we can discover the effect of stretching and jamming on the string recognition time for strings of different length.

10. Presentation of results

We will mainly use graphs to present our results. In the graphs for memory usage measurements we will set out the memory usage for different stretch and jam factors against the transition table density. For the measurement of string recognition time, we will set out the string recognition time against different stretch and jam factors.

In the list of steps above we spoke of using random generated DFAs and random generated paths over DFAs. In the next two sections we will present algorithms for these tasks. To be able to choose a random alphabet symbol from a set of symbols and a random state from a set of states etc. we make use of a random number generator. A good random number generator is essential for obtaining reliable results. There is a certain list of statistical tests that any good random number generator ought to pass [PTVF92]. In our algorithms we will exclusively make use of random number generator ran1 from [PTVF92], this generator is known to pass all statistical tests. ran1 is a function that returns a uniform random deviate between 0.0 and 1.0. The main advantage of using a function is that in case of unexpected behavior, the same random sequence can be generated multiple times.

We also spoke of measuring the transition table size. Here, our use of the STL in combination with C++ proved to be a problem. Our transition table is basically a 2-dimensional STL vector. Memory management for vectors is handled by the STL itself, and vectors grow and shrink as needed. There are ways to influence the memory management but it remains difficult to get reliable results about the memory usage. Specifically, the usage of the C++ sizeof() operator might give us the same value for vectors of different length because the STL reserved the same amount of memory for them.

Therefore, we have chosen to calculate the memory usage ourself. The minimal amount of memory a transition table uses, is the sum of the amount of memory over all cells. In DFAs, cells represent states, so the memory usage for each cell is determined by the memory usage to represent a state, which is in turn determined by the total number of states. For a DFA with a set of states \(|Q|\) the amount of memory to represent a state is \(\lceil 2\log(|Q|) \rceil\) bits.

The number of cells of a transition table is equal to the number of rows multiplied by the number of columns. In other words, the number of cells is equal to the number of states multiplied by the number of alphabet symbols. Therefore, for a DFA with a set of states \(Q\) and alphabet \(\Sigma\), the calculated memory usage is: \(|Q| \cdot |\Sigma| \cdot \lceil 2\log(|Q|) \rceil\) bits. This is the absolute minimum amount of memory needed for a transition table representation, and thus the most reliable way to compare the memory usage for different transition tables.

To be able to perform accurate timing measurements, we need a reliable timer mechanism. The C++ standard library provides a function clock(), but this function is not suited for the accurate timing measurements we want to perform. Some processors, like the x86 family of processors and the PowerPC processors have a special register, called the time stamp or time base register. This register keeps an accurate count of every cycle that occurs on the processor. We can get an accurate value of the number of clock cycles that have passed during execution of a piece of code if we read the register immediately before and after the piece of
code. This can be done on x86 processors by the RDTSC() instruction, which is described in [Int97]. On Mac OS X systems, which use the PowerPC processor, this can be done by using the CHUD Toolset. For an overview of the CHUD Toolset, see [App04].

Below it is shown how we use the timing mechanism in the case of an 8-bit DFA and, after stretching by a factor 2, by a 4-bit DFA. We do not use the public NextState methods of the classes because these methods might cause a little overhead. PERFSTART and PERFSTOP are macros that read the time stamp register and write it to two 64-bit variable. The difference of these variables is equal to the number of clock cycles passed.

We use an array of character bytes to store the symbols of the string. So, in case of a 1-bit DFA, we fill every byte with 8 symbols. In case of an 8-bit DFA we take every byte as a single input symbol.

Of course, this means that we do some bit shifting and masking to retrieve the right symbol if the symbol size is smaller than 8 bits. If we want to avoid bit shifting and masking we need to represent every symbol with a single byte. It could be faster to process strings this way.

However, if we represent every symbol with a single byte, the input has to stretched too. For example, a 1-bit symbol will not use one bit of a byte but it will use a complete byte. We call this pre-stretching the input string, and it will not be discussed in this thesis. We leave this as future work.

The input string, together with the transition table and a variable to store the state, tmpState, can be used to measure the time it takes for the transition table to process the string. The first piece of code shows the timing of the original DFA. The second piece of code shows the timing for the 2-stretch of the original DFA. In that case, each byte is processed in two steps of 4 bits.

PRESTART
```c
  tmpState=startState;
  for ( i = 0; i<pathLen; i++) {
    tmpState=TransitionTable[tmpState][path[i]];
  }
PRESTOP
```

PRESTART
```c
  tmpState=startState;
  for ( i = 0; i<pathLen; i++) {
    tmpState=TransitionTable[tmpState][path[i] & 0xf];
    tmpState=TransitionTable[tmpState][((path[i]>>4) & 0xf];
  }
PRESTOP
```

### 7.1 Generation of Random DFAs

For benchmarking purposes we need DFAs that we can use as input for our algorithms. One way to do this is to create random DFAs. To be able to stretch a DFA FA₀ with algorithm 5.3, all states in FA₀ need to be reachable from the start state. Also, from all states there must be a path to at least one final state.

This leads us to construct a random DFA generator in the following way: The random DFA generator takes a number of states Q, an alphabet Σ, a start state q a set of final states F and
the desired transition density \( D \) as input. To make every state reachable from the start state we divide all states into two sets \( R \) and \( S \). We maintain invariant \((R \cup S = Q) \land (R \cap S = \emptyset)\). Initially \( R = \{q\} \) and \( S = Q \setminus \{q\} \). Then a random transition between a state \( r \in R \) and a state \( s \in S \) is created. Next, \( s \) is removed from \( S \) and added to \( R \). This process can be repeated until \( S = \emptyset \).

The same procedure can be used to create a path for each state to at least one final state. Because every final state is already reachable from the start state after the previous procedure, a number of states already has a path to a final state.

After these two procedures, random transitions can be added to the DFA until density \( D \) is reached.

**Proposition 7.1.** Algorithm 7.2 creates a random DFA \( \mathcal{F}A_0 \) with a set of states \( Q \), alphabet \( \Sigma \), start state \( q \), a set of final states \( F \) and a transition density at least \( D \).

**Algorithm 7.2** RANDOMDFA\((\mathcal{F}A_0, Q, \Sigma, q, F, D)\):

\[
\begin{align*}
\text{Pre:} & \\
\text{Post:} & \\
|Q_0| := Q & ; \Sigma_0 := \Sigma \\
q_0 := q & ; F_0 := F \\
t := 0 & \\
R, S := \{q\} \setminus R & ; G, H := F, S \setminus G \\
P[0 \ldots |Q| - 1] := \emptyset & \\
\text{do } S \neq \emptyset & \\
\quad p := \text{Random}(R); a := \text{Random}(\Sigma); q := \text{Random}(S) & \\
\quad ; \delta_0 := \delta_0 \cup \{(p, a, q)\} & \\
\quad ; t := t + 1 & \\
\quad ; P[q] := p & \\
\quad ; R, S := R \cup q, S \setminus q & \\
\text{od} & \\
\text{for all } g \in G & \\
\quad s := P[g] & \\
\quad ; \text{do } s \neq \emptyset & \\
\quad \quad H := H \setminus s & \\
\quad \text{od} & \\
\text{rof} & \\
\text{do } H \neq \emptyset & \\
\quad p := \text{Random}(G); a := \text{Random}(\Sigma); q := \text{Random}(H) & \\
\quad ; \delta_0 := \delta_0 \cup \{(q, a, p)\} & \\
\quad ; t := t + 1 & \\
\quad ; G, H := G \cup q, H \setminus q & \\
\text{od} & \\
\text{do } \frac{t}{|Q| \times 100} < D & \\
\quad p := \text{Random}(Q); a := \text{Random}(\Sigma); q := \text{Random}(Q) & \\
\end{align*}
\]
if $\delta_0(p, q) = \emptyset \rightarrow$

\[
\delta_0 := \delta_0 \cup \{((p, a), q)\}
\]

\[
; t := t + 1
\]

fi

Proof of proposition 7.1. Proof omitted.

This algorithm can easily be adapted to create random NFAs.

### 7.2 Generation of Random Paths

To be able to measure the string recognition time in our benchmarking tests we want to measure the time a DFA needs to process a number of sequential transitions. For this purpose we want to generate random paths of transitions for a DFA. Algorithm 7.4 below returns the label of a random path with a specific start state and length for a DFA. It is based on the DFS algorithm, see chapter 5. Algorithm RANDOMPATH calls algorithm RANDOMPATH-DFS. RANDOMPATH-DFS descends recursively into the DFA to find random paths of increasing length until the specified length has been reached.

**Proposition 7.3.** Algorithm 7.4 generates a random path of transitions in DFA $FA_0$ with start state $q$ and length $f$.

**Algorithm 7.4 RANDOMPATH($FA_0$, $q$, $f$):**

\[
\begin{align*}
&\{ l := \epsilon; \\
&\; m := \epsilon; \text{RANDOMPATH-DFS($FA_0, q, f, 0, l, m, \text{true}$)}; \\
&\; \text{return } m
\end{align*}
\]

**Algorithm 7.5 RANDOMPATH-DFS($FA_0$, $p$, $d$, $c$, $l$, $m$, $b$):**

\[
\begin{align*}
&\{ ; S := \emptyset \\
&\; \text{if } c < d \rightarrow \\
&\; \quad \text{for all } q, a : ((p, a), q) \in \delta_0 \rightarrow \\
&\; \quad S := S \cup a \\
&\; \quad \text{rof} \\
&\; \quad \text{do } S \neq \emptyset \land b \\
&\; \quad \quad a := \text{Random($S$)}; \\
&\; \quad \quad b := \text{RANDOMPATH-DFS($FA_0, \delta_0(p, a), d, c + 1, l a, m$)}; \\
&\; \quad \quad S := S \setminus s; \\
&\; \quad \text{od} \\
&\; \quad c = d \rightarrow \\
&\; \quad m := l \\
&\; \quad \text{return } \text{false} \\
&\; \text{fi}
\end{align*}
\]
7.3 Performance Results for Stretching

In this section we will present the results of the performance evaluation of the stretching operation. We will first present the performance results for memory usage. After that, we will present the performance results for processing strings.

As indicated in section 7.0, the workload for the memory usage experiments consists of random generated DFAs with a specific alphabet size, number of states and transition density. Subsequently, these DFAs are stretched by various factors and the memory usage of the transition tables will be compared.

For the first two benchmarks, we generated random 2-bit DFAs with increasing transition density. Here, we were faced with a problem. A 2-bit DFA has a transition table with 4 columns for the alphabet symbols. From every state there must be a path to at least one final state, therefore every state (except the final states) must have at least one out-going transition. This means that for most rows in the transition table, at least one of the four cells is filled. As a result, random generated 2-bit DFAs have a density of at least 25 percent. In fact, in practice our random generated 2-bit DFAs have a density of at least 27 percent, therefore we used 27 percent transition density as a starting point.

Figures 7.0 and 7.1 show the results for stretching 2-bit DFAs of 100 and 1000 states, respectively. Of course, a 2-bit DFA can only be stretched by a factor 2, so we only show the memory usage for the DFAs before and after stretching by a factor 2. To get these results we generated a random DFA, stretched it by a factor 2 and calculated the memory usage for both DFAs. We repeated these experiments 100 times and calculated the arithmetic mean over these 100 experiments to obtain reliable results.

The solid horizontal line in both figures indicates the amount of memory (in bytes) the original DFA uses. In the case of a 2-bit DFA with 100 states, the amount of memory is equal to \(100 \times 2^2 \times \lceil 2\log(100) \rceil = 2800\) bits, or 350 bytes.

As we have seen in chapter 3, the memory usage of the stretched DFA depends on the number of intermediate states that are introduced, which in turn depends on the alphabet size, stretch factor and the transition density. The average memory usage for the 100 stretched DFAs is indicated in both figures by the dashed line. To give an indication of the variance in the results we use a gray range sweep. The bottom of the range sweep indicates the lowest memory usage and the top the highest memory usage we recorded out of the 100 stretched DFAs.

From these figures we cannot confirm the assumption of chapter 3 that stretching reduces the memory usage if the transition density is low, simply because 2-bit DFAs cannot have a low transition density. We did not take this into account for our theoretical results. We can however confirm the assumption that the increase or decrease in memory usage is independent of the number of states, because in both figures we can see a similar increase in memory usage.

Furthermore, we can conclude that the variance in the memory usage is very small. In the first figure, the minimum memory usage is around 10 bytes less than the average and the maximum memory usage is around 10 bytes more. In the second figure, this variance is around 100 bytes on both sides of the average.
Figure 7.0: Memory usage for random 2-bit DFAs of 100 states

Figure 7.1: Memory usage for random 2-bit DFAs of 1000 states
The next three figures, figures 7.2, 7.3 and 7.4, deal with 4-bit DFAs. For this experiment, we stretched a 4-bit DFA by factors 2 and 4 respectively, and repeated this experiment also 100 times.

Because the number of columns in a transition table of a 4-bit DFA is 16, these DFAs have a transition density of at least 7 percent. In these figures we see that between 15 and 20 percent transition density, the average memory usage after stretching by a factor 2 becomes higher than that of the original DFA. In other words, the break-even point for stretching by a factor 2 lies between 15 and 20 percent. This is true for DFAs of 10 states, 100 states and 1000 states, although increasing the number of states shifts the break-even point a little to the right. This means that a higher number of states has a positive influence on stretching. Therefore, we can conclude that if a 4-bit DFA has a transition density lower than 15 percent, stretching by a factor 2 will decrease the memory usage.

For stretching by a factor 4, this point lies between 10 and 15 percent. This confirms the assumptions we made in chapter 3. Figure 3.6 in chapter 3 shows the break-even point for stretching by a factor 2, 4 and 8. From this figure we can draw the conclusion that in the case of 4-bit DFAs, which have 16 alphabet symbols, the break-even point lies between 15 and 20 percent for stretching by a factor 2, and between 10 and 15 percent for stretching by a factor 4. From the experiments we can also conclude that the variance of results is increasing in comparison with the variance in the results for 2-bit DFAs.

The next four figures 7.5, 7.6, 7.7 and 7.8 present the results for 8-bit DFAs. For these DFAs we performed the same type of experiments as for the 2 and 4-bit DFAs. In this case we stretched by a factor 2, 4 and 8. These experiments also confirm the expectations we made in chapter 3.

Figure 7.2: Memory usage for random 4-bit DFAs of 10 states
Figure 7.3: Memory usage for random 4-bit DFAs of 100 states

Figure 7.4: Memory usage for random 4-bit DFAs of 1000 states
We performed one additional benchmark for the 8-bit case. In chapter 3 we indicated that we can stretch a transition with either the least significant bit first, or with the most significant bit first. In these experiments we stretched with the least significant bit first, but it is also possible to take the other option. This might have effect on the memory usage because it can influence the determinization and minimization process of the stretching algorithm.

In figure 7.8 we see the results of stretching every transition with the most significant bit first. We can see that the results are almost identical to the previous case. There are two simple reasons for this. Firstly, as we indicated before, the random number generator we use is a function that generates a uniform distribution of numbers. Because we use the same initialization value for both cases, it generates the same sequence of numbers so the experiments are performed on exactly the same set of random DFAs.

Secondly, because the transition density is very low, below 15%, most states will not have many outgoing or incoming transitions. Therefore, the determinization and minimization process will not reduce the number of states very much.

To measure the string processing time we performed a set of different tests. We performed all tests on an 800 MHz AMD Athlon system under Debian Linux 3.0, kernel version 2.4.22. The compiler we used is GCC 3.3.2. To minimize interfering processes all tests were performed at runlevel 1 (single user mode).

For the first test, we generated 10 random 8-bit DFAs with 100 states and a transition density of 10 percent. For each DFA we generated a random path of 8 bytes which served as input string for the DFA. We stretched each DFA by factor 2, 4 and 8 which resulted in 4-bit, 2-bit and 1-bit DFAs respectively. After that, every DFA processed the input string 100 times and we calculated the average time it took to process the string. We repeated this experiment for strings of 16 bytes and 32 bytes.
Figure 7.6: Memory usage for random 8-bit DFAs of 100 states

Figure 7.7: Memory usage for random 8-bit DFAs of 1000 states
The results of this test are presented in figure 7.9. The figure shows three groups of bars to indicate the time to process a string of 8 bytes, 16 bytes and 32 bytes respectively. Each group of bars contains four bars, for the original DFA and the stretched DFAs. We also calculated the standard deviation over all individual tests. The standard deviation is indicated by the thin line with horizontal tails.

We can see that for an 8-bit DFA, the time to process 8 bytes is 0.53 microseconds. After stretching by a factor 2 this time doubles to 1.07 microseconds. The time almost quadruples after stretching by a factor 4. And after stretching by a factor 8, the time to process 8 bytes is almost 6 times as long as for the original DFA.

For 16-byte strings, the increase in processing time is slightly higher than for 8-byte strings. This holds for stretching by a factor 2 as well as factor 4 and 8. For 32-byte strings the increase is higher compared to 16-byte strings. Therefore, the processing time after stretching is relatively higher for longer string lengths.

In figure 7.10 we present the results for our second timing benchmark. In this test we generated again 8-bit DFAs with 100 states, but this time with a transition density of 50 percent. The results are very similar to the results from the previous test. From these results we conclude that the transition density has no influence on the string processing time before or after stretching. This is in line with what we expected.

In figure 7.11 we present the results for the next benchmark test. The test is the same as the first timing benchmark, the only exception being DFAs with 1000 states instead of 100. We expected the number of states to have no influence on the string recognition time but this is not the case. Although the results are very similar the string processing time is slightly
Figure 7.9: String processing time for random 8-bit DFAs of 100 states, 10% density

Figure 7.10: String processing time for random 8-bit DFAs of 100 states, 50% density
higher for each individual test. This is even more apparent for larger strings. We also notice an increase in the standard deviation in each case.

We also performed benchmark tests on 4-bit DFAs. The results are presented in figure 7.12. Because a 4-bit DFA cannot be stretched by a factor 8, every group of bars only contains 3 bars. These 3 bars can be compared to the last 3 bars in each group of the first experiment because in both experiments these bars represent 4-bit, 2-bit and 1-bit DFAs.

We see that the processing time in this test is lower than for the same DFAs in the first test. What we have to take into account however, is that the 4-bit DFAs of this experiment have 100 states and a transition density of 10 percent. In the first timing experiment, the original 8-bit DFA also has 100 states and a transition density of 10 percent. But after stretching by a factor 2, the resulting 4-bit DFA will have much more than 100 states and a transition density higher that 10 percent. We have seen in previous tests that the number of states has influence on the string processing time. This can partially explain the difference in the results.

We also see that the standard deviation for each case is much smaller than in the first experiment.

### 7.4 Performance Results for Jamming

For the performance evaluation of the jamming operation we took the same approach as for stretching. We will first present the results for memory usage, and subsequently the results for string processing time.
<table>
<thead>
<tr>
<th>String Length (bytes)</th>
<th>Time (µsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.89</td>
</tr>
<tr>
<td>16</td>
<td>1.60</td>
</tr>
<tr>
<td>32</td>
<td>2.93</td>
</tr>
</tbody>
</table>

Figure 7.12: String processing time for random 4-bit DFAs of 100 states, 10% density

<table>
<thead>
<tr>
<th>Memory usage for random 1-bit DFAs of 100 states</th>
</tr>
</thead>
<tbody>
<tr>
<td>Density (%)</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>70</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>

Figure 7.13: Memory usage for random 1-bit DFAs of 100 states
For the first two benchmarks we jammed 100 random 1-bit DFAs successively by a factor 2, 4 and 8. The results are presented in figures 7.13 and 7.14. Note that, to be able to include results for factor 8, there is a break in the vertical axis. After this break we continue on a larger scale to be able to present everything in one figure. This might be a bit misleading at first. The presentation is identical to the presentation of memory usage for stretching in the previous section. A solid horizontal line indicates the memory usage for the original 1-bit DFA. The dashed lines are used for the average memory usage of the DFAs after jamming by a factor 2, 4 and 8 respectively. Again, the gray range sweep indicates the range for the memory usage out of 100 jammed DFAs.

What stands out immediately is the horizontal upper bound in the range of memory usage for all three factors of jamming. There is a simple explanation for this. Jamming a DFA has the potential to reduce the number of states, but in the worst case there are no redundant states and the number of states after jamming is equal to the number of states before jamming. Because the new alphabet size is fixed by the original alphabet size and the factor of jamming, for each DFA there is a specific upper bound in memory usage after jamming. We can conclude from these figures that in the case of 100 random DFAs there is always at least one DFA that has no redundant states.

Another conclusion we can draw from these figures is that when the transition density is increased the average memory usage approaches the upper bound. This is not a surprise because the chance that a state is redundant decreases when the number of transitions increases.

From our results for the 1-bit DFAs with 100 states we conclude that the upper bound
Figure 7.15: Memory usage for random 2-bit DFAs of 100 states

Figure 7.16: Memory usage for random 2-bit DFAs of 1000 states
for memory usage after jamming by factor 2 lies at 379 bytes, which is approximately twice as much as the 175 bytes that are needed for the original DFA. For the case of factor 4, this is 1591 bytes, approximately 9 times as much. The upper bound for memory usage after jamming by a factor 8 is 25831 bytes, that is approximately 148 times as much as the memory needed for the transition table of the original DFA. For the 1-bit DFAs with 1000 states the increase in memory usage is the same.

In figures 7.15 and 7.16, we present the results for jamming 2-bit DFAs. From the results we can draw the conclusion that the increase in memory usage after jamming by a factor 2 is approximately 4 times as much. For the other factors, the increase is even greater. The pattern is clear: the larger the alphabet size and the jam factor, the higher the memory usage for the jammed DFA becomes.

The results for jamming 4-bit DFAs are presented in figures 7.17 and 7.18. The figures speak for themselves.

The last set of benchmark tests were performed to measure the string processing time for different jamming scenarios. These tests were carried out the same way as the timing experiments for stretching.

For the first test, we generated 10 random 1-bit DFAs with 100 states and a transition density of 70 percent. For each DFA we generated a random path of 8 bytes which served as input string for the DFA. We jammed each DFA by factor 2, 4 and 8 which resulted in 2-bit, 4-bit and 8-bit DFAs respectively. After that, every DFA processed the input string 100 times and we calculated the average time it took to process the string as well as the standard deviation. We repeated this experiment for strings of 16 bytes and 32 bytes.
Figure 7.18: Memory usage for random 4-bit DFAs of 100 states

Figure 7.19: Memory usage for random 1-bit DFAs of 100 states, 70% density
String processing time for random 1-bit DFAs of 1000 states, 70% density

<table>
<thead>
<tr>
<th>String Length (bytes)</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (µsec)</td>
<td>2.09</td>
<td>3.94</td>
<td>7.68</td>
</tr>
<tr>
<td>2−Jam</td>
<td>2.05</td>
<td>3.97</td>
<td>7.76</td>
</tr>
<tr>
<td>4−Jam</td>
<td>1.00</td>
<td>1.91</td>
<td>3.70</td>
</tr>
<tr>
<td>8−Jam</td>
<td>0.45</td>
<td>0.84</td>
<td>1.50</td>
</tr>
</tbody>
</table>

Figure 7.20: Memory usage for random 1-bit DFAs of 1000 states, 70% density

In figure 7.19 we present the results for the first test. We see that jamming a 1-bit DFA by a factor 2 does not influence the time to process a string. Jamming by a factor 4 does, however. The time to process a string of 8 bytes as well as 16 and 32 bytes halves after the DFA is jammed by a factor 4.

After jamming by a factor 8, the time to process an 8-byte string is less than a quarter than than for the original DFA. And the time to process a 16-byte or 32-byte string after jamming by a factor 8 takes even less than one fifth of the time needed with the original DFA.

Figure 7.20 presents the results for the second timing benchmark for jamming. We see that the time to process the strings is slightly higher than in the first test. We saw the same pattern in the timing benchmarks for stretching. The relative proportions in the timing after jamming by a factor 2, 4 and 8 are the same as in the first test.

The results of the last benchmark we performed is presented in figure 7.21. For this test we used 2-bit DFAs instead of 1-bit DFAs. The three groups of bars for the different string lengths can be compared to the last three bars of each group of the first timing benchmark.

The time to process the strings is almost equivalent to the time in the first test except for the first bar in each group. In these cases the time is a little less in this last test, as was the case with stretching. The relative proportions in the timing after jamming by a factor 2, 4 and 8 are also a smaller than in the first test. From this we can conclude that for a larger alphabet size, the advantage in string processing time after jamming is smaller.

This concludes the presentation of our results.
String processing time for random 2-bit DFAs of 100 states, 70% density

<table>
<thead>
<tr>
<th>String Length (bytes)</th>
<th>Time (µsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1.34</td>
</tr>
<tr>
<td>16</td>
<td>2.57</td>
</tr>
<tr>
<td>32</td>
<td>5.03</td>
</tr>
<tr>
<td>8</td>
<td>0.97</td>
</tr>
<tr>
<td>16</td>
<td>1.78</td>
</tr>
<tr>
<td>32</td>
<td>3.45</td>
</tr>
<tr>
<td>8</td>
<td>0.97</td>
</tr>
<tr>
<td>16</td>
<td>1.78</td>
</tr>
<tr>
<td>32</td>
<td>3.45</td>
</tr>
</tbody>
</table>

Original

Figure 7.21: Memory usage for random 2-bit DFAs of 100 states, 70% density

7.5 Conclusions

In this section we will summarize the conclusions we found in our performance evaluation of stretching and jamming DFAs. Stretching a DFA will introduce new states for each original transition, therefore the transition density of a transition table is a crucial factor in determining the usefulness of stretching. If the transition density is high, the number of introduced states is high and therefore the memory usage increases. In the case the transition density is low, the opposite is true.

For most 2-bit DFAs, stretching is not useful because 2-bit DFAs almost always have a high transition density. Stretching 4-bit DFAs by a factor 2 into 2-bit DFAs will decrease the memory usage if the transition density is between 15 and 20 percent. The stretching operation is less efficient for stretching by a factor 4.

In the case of 8-bit DFAs, stretching by a factor 4 into 2-bit DFAs is the most efficient transformation. It reduces the memory usage if the transition density is lower than around 9 percent. This indicates that 2-bit DFAs are the most memory efficient DFAs but more practical results must be gathered to verify that.

The number of states has a small influence on the memory usage after stretching. We can conclude that increasing the number of states increases the usefulness of stretching. This influence is very minor though.

The variance in the results is not very high in most cases. This indicates that in the general case, if we know the transition density of a DFA, we can accurately predict the memory usage after stretching.
Stretching has a negative effect on the string processing time. After stretching an 8-bit DFA by a factor 2, the string processing time for 8-byte strings doubles. It almost quadruples after stretching by a factor 4. After stretching by a factor 8 the string processing time is almost 6 times as long as for the original DFA.

The performance figures are slightly worse for longer strings. Transition density does not have any influence on the string processing time but the number of states does. Increasing the number of states has a negative effect on the performance after stretching.

Jamming has a negative effect on the memory usage but a positive effect on the string processing time. The variance in the memory usage is high but there is a clear upper bound after jamming. The reason for this is that the number of states is bounded from above because in the worst case the number of states after jamming is equal to the number of states of the original DFA plus one special bottom state. The new alphabet size is fixed by the original alphabet size and the jam factor. Because the memory usage of the transition table is only dependent on the number of states and the alphabet size, the memory usage is bounded from above.

Furthermore, we can conclude that the number of states does not have a big influence on the memory usage after jamming. Increasing the alphabet size and of course increasing the jam factor does have a negative effect on jamming.

We do see a positive effect on the string processing time after jamming though. For 1-bit DFAs the string processing time halves after jamming by a factor 4. Jamming by a factor 8 results in a string processing time that is less than a quarter of the original time. Increasing the string length has a positive effect on jamming. Changing the number of states only has a minor influence.

This almost ends our discussion of the performance results for both stretching and jamming. We will end with a note about the reliability of our results and indications for future work to gather more benchmarking data.

Due to limited time and resources, the results we gathered are only for relatively small DFAs. The largest alphabet size we used was 8 bits, or 256 symbols. We did not consider DFAs that have more than 1000 states.

Also, every benchmarking study is influenced by a number of factors such as implementation details, the platform software and hardware and so on. Therefore, more practical results must be gathered with different implementation languages, compilers, operating systems, hardware platforms etc.. Besides that, the usefulness of stretching and jamming must be tested for larger DFAs, and possibly different transition table implementations.
Chapter 8

Conclusions and Future Work

The most important conclusions can be found in section 7.5. However, in this chapter we will present some general conclusions we can draw from this thesis.

We have defined the notions of stretching and jamming and shown the theoretical conditions under which they influence performance. In the case of stretching, performance can be improved by reducing the memory usage. Jamming on the other hand, increases memory usage. During the theoretical discussion of jamming we hinted that it might reduce the string processing time but we did not present a theoretical model for that.

We we did empirical research by benchmarking the stretching and jamming operations under different conditions. The benchmarking data we gathered backs up our claim that stretching can reduce memory usage when the transition density is small, i.e. smaller than 15 percent. This comes at the price of increased string processing time.

In the case of jamming we saw, as expected, that the roles are reversed. Our benchmarking data shows that jamming improves performance by reducing the string recognition time. This time however, it comes at the price of increased memory usage. We can conclude that jamming is most useful in cases where the alphabet size is small, i.e. smaller than 256 symbols.

There are a number of interesting problems that can still be investigated further. We only considered stretching or jamming the complete transition table. Transforming only a small part of the transition table, in other words local stretching and jamming, is an interesting problem for further research.

Of course, any method that is used to change the transition table, for example character classes and sparse matrices, influences stretching and jamming. Therefore, these situations can be investigated further.

Lastly, we only looked into transforming automata and not regular expressions. The stretching and jamming of regular expressions is also a candidate for further research.

This concludes our discussion of stretching and jamming and forms the end of this thesis.
Appendix A

Source code

Algorithm A.1. MinDStretch(StretchFactor stretchFactor, bool reverse) const
This algorithm implements algorithm 5.3. It also performs the minimization step as proposed in algorithm 5.5.

```cpp
DFA *DFA::MinDStretch(StretchFactor stretchFactor, bool reverse) const
{
    Assert<invalid_argument>(AlphabetBitSize() % stretchFactor == 0,
        invalid_argument("Invalid argument for function DStretch.
        AlphabetBitSize() % stretchFactor != 0"));

    State inputState = 0;
    AlphabetSize newAlphabetSize = (AlphabetSize)(AlphabetBitSize() / stretchFactor);
    queue<State> Q;
    Q.push(startState_);
    vector<State> V(NumStates(), 0);
    V[startState_] = 1;

    DFA *stretchedDFA = new DFA(NumStates(), newAlphabetSize, startState_, finalStates_);

    stretchedDFA->IsStretched(true);
    stretchedDFA->IsJammed(false);
    stretchedDFA->Factor(stretchFactor);

    while (!Q.empty()) {
        inputState = Q.front();
        Q.pop();
        MinDStretchState(stretchedDFA, Q, V, inputState, stretchFactor, reverse);
    }

    return stretchedDFA;
}
```
Algorithm A.2. MinDStretchState(DFA *stretchedDF A, queue<State> &Q, vector<State> &V, State startState, StretchFactor stretchFactor, bool reverse) const

This algorithm is used by algorithm A.1 to implement algorithm 5.3.

```cpp
void DFA::MinDStretchState(DFA *stretchedDF A, queue<State> &Q, vector<State> &V, State startState, StretchFactor stretchFactor, bool reverse) const
{
    State nextState = 0, newState = stretchedDF A->NumStates(), tmpState = 0, i = 0;
    AlphabetSize tmpSymbol = 0, newAlphabetSize = (AlphabetSize)(AlphabetBitSize() / stretchFactor);
    vector< set<State> > intermediateStates(stretchFactor, set<State>());
    intermediateStates[0].insert(startState);
    for (AlphabetSize inputSymbol = 0; inputSymbol < AlphabetDecSize(); inputSymbol++) {
        nextState = (*transitionTable_.)[startState][inputSymbol];
        if (nextState != EMPTY_STATE) {
            i = 0;
            if (reverse) tmpSymbol = Reverse(inputSymbol, AlphabetBitSize());
            else tmpSymbol = inputSymbol;
            tmpState = startState;
            while (((stretchedDF A->transitionTable_))[tmpState][BitMask(tmpSymbol, newAlphabetSize)]) != EMPTY_STATE) {
                tmpState = (((stretchedDF A->transitionTable_))[tmpState][BitMask(tmpSymbol, newAlphabetSize)];
                tmpSymbol = tmpSymbol >> newAlphabetSize;
                i++;
            }
            stretchedDF A->Resize(max(stretchedDF A->NumStates(), (State)(stretchedDF A->NumStates() + stretchFactor - i - 1))) ;
            for (StretchFactor f = i; f < stretchFactor - 1; f++) {
                stretchedDF A->AddTransition(tmpState, BitMask(tmpSymbol, newAlphabetSize), newState);
                intermediateStates[f + 1].insert(newState);
                tmpSymbol = tmpSymbol >> newAlphabetSize;
                tmpState = newState;
                newState++;
            }
            stretchedDF A->AddTransition(tmpState, BitMask(tmpSymbol, newAlphabetSize), nextState);
        } else {
            Q.push(nextState);
            V[nextState] = 1;
        }
    }
}
```
stretchedDFA->MinimizeStretch(intermediateStates, stretchFactor);
Algorithm A.3. MinimizeStretch(const vector<set<State>>& intermediateStates, StretchFactor stretchFactor)
This algorithm implements algorithm 5.5. It is used by algorithm A.1 to implement algorithm 5.3.

```cpp
void DFA::MinimizeStretch(const vector<set<State>>& intermediateStates, StretchFactor stretchFactor)
{
    bool eq;
    set<State> deleteStates;

    for (StretchFactor f=stretchFactor; f>0; f--) {
        for (set<State>::iterator xIt = intermediateStates[f-1].begin(); xIt != intermediateStates[f-1].end(); xIt++) {
            for (set<State>::iterator yIt = xIt; yIt != intermediateStates[f-1].end(); yIt++) {
                if (*xIt != *yIt) {
                    eq=true;
                    for (AlphabetSize inputSymbol=0; inputSymbol<AlphabetDecSize(); inputSymbol++) {
                        if ( (*transitionTable_)[*xIt][inputSymbol] != (*transitionTable_)[*yIt][inputSymbol] ) eq = false;
                    }
                    if (eq) {
                        deleteStates.insert(*yIt);
                        for (set<State>::iterator setIt=
                            intermediateStates[f-2].begin(); setIt!=
                            intermediateStates[f-2].end(); setIt++) {
                            if ( ( (*transitionTable_)[*setIt][
                                inputSymbol] == *yIt ) ( *transitionTable_)[*setIt][inputSymbol]
                            ==*xIt ;
                        }
                    }
                }
            }
        }
    }
    DeleteStates(deleteStates);
}
```
This algorithm is used by algorithm A.3 to implement algorithm 5.5.

```cpp
void DFA::DeleteStates(const set<State>& deleteStates)
{
    State s = 0;
    vector<State> states(NumStates(), State());

    for (set<State>::iterator setIt=deleteStates.begin(); setIt!=
         deleteStates.end(); setIt++) states[*setIt] = 1;
    for (Vec<State, State>::VecIterator vecIt=states.begin(); vecIt!=
         states.end(); vecIt++) {
        if (*vecIt == 1) s++;
        *vecIt = s;
    }
    s = 0;

    for (TransitionTable<State>::RowIterator rowIt=transitionTable_\_->
         begin(); rowIt!=transitionTable_\_->end(); rowIt++) {
        if (deleteStates.count(s) != 0) {
            transitionTable_\_->erase(rowIt);
            rowIt --;
        } else {
            for (TransitionTable<State>::ColIterator colIt=rowIt->begin()
                 ; colIt!=rowIt->end(); colIt++) if (*colIt != EMPTY_STATE)
            {
                *colIt = states[*colIt];
            }
            s++;
        }
    }
}
```
Algorithm A.5. DStretch(const StretchFactor stretchFactor, bool reverse) const

This algorithm implements algorithm 5.3.

```cpp
NFA *NFA::DStretch(const StretchFactor stretchFactor, bool reverse) const
{
    Assert<invalid_argument>(AlphabetBitSize() % stretchFactor == 0,
        invalid_argument("Invalid argument for function DStretch.
            AlphabetBitSize() % stretchFactor != 0"));

    State nextState = 0, inputState = 0, newState = NumStates(), tmpState = 0, i = 0;
    AlphabetSize tmpSymbol = 0, maskSymbol = 0, newAlphabetSize =
        (AlphabetSize)(AlphabetBitSize() / stretchFactor);
    queue<State> Q;
    vector<State> V(NumStates(), 0);
    V[startState] = 1;

    NFA *stretchedNFA = new NFA(NumStates(), newAlphabetSize, startState_,
        finalStates_);

    while (!Q.empty()) {
        inputState = Q.front();
        Q.pop();

        for (AlphabetSize inputSymbol = 0; inputSymbol < AlphabetDecSize();
            inputSymbol++) {
            for (set<State>::iterator setIt = ((transitionTable_)[
                inputState][inputSymbol]).begin(); setIt != ((
                transitionTable_)[inputState][inputSymbol]).end(); setIt++) {
                i = 0;

                nextState = *setIt;

                if (reverse) tmpSymbol = Reverse(inputSymbol, AlphabetBitSize());
                else tmpSymbol = inputSymbol;

                maskSymbol = BitMask(tmpSymbol, newAlphabetSize);

                tmpState = inputState;

                stretchedNFA->Print();

                if (!(((**stretchedNFA)[tmpState][maskSymbol]).empty())) {
                    while (*(((**stretchedNFA)[tmpState][maskSymbol]).
                        begin()) >= this->NumStates()) {
                        tmpState = *(((**stretchedNFA)[tmpState][
                            maskSymbol]).begin());
                        tmpSymbol = tmpSymbol >> newAlphabetSize;

                        maskSymbol = BitMask(tmpSymbol, newAlphabetSize);
                        i++;
```
stretchedNFA->Resize(max(stretchedNFA->NumStates(), (State)(stretchedNFA->NumStates() + stretchFactor - i - 1)))
;

for (StretchFactor f = 0; f < stretchFactor - i - 1; ++f) {
    stretchedNFA->AddTransition(tmpState, BitMask(tmpSymbol, newAlphabetSize), newState);
    tmpSymbol = tmpSymbol >> newAlphabetSize;
    ++tmpState = newState;
}

stretchedNFA->AddTransition(tmpState, BitMask(tmpSymbol, newAlphabetSize), nextState);

if (V[nextState] == 0) {
    Q.push(nextState);
    V[nextState] = 1;
}

return stretchedNFA;
Bibliography


