Reducing Nondeterministic Finite Automata

Alice Pavaux

Erasmus (Mathematics)

2011/2012

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)
Summary

A finite automaton is an abstract computing machine, containing a set of “states”, and describing a language. At any time, the automaton is in a fixed state, and an input event corresponding to a symbol makes it switch to another state. Nondeterminism allows these machines to be in several states at once, although this does not make them more powerful: the languages recognized are the regular languages in both deterministic and nondeterministic cases. Still nondeterministic finite automata (NFAs) are interesting, because for a given language they can be smaller than the minimum deterministic finite automaton describing the same language, and this could save some execution time for the applications using them. This project aims to explore ways of reducing the size of NFAs. After research into possible techniques, an implementation work was done to apply these techniques and to help evaluate their actual efficiency.
Acknowledgements

First I would like to thank Matthew Hubbard for his help and advice at the beginning of the year, as I was worrying about the project, and for the suggestions he made of potential supervisors according to my fields of interest. I am very grateful to Haiko Müller, my supervisor, for the proposal of this interesting subject as well as his support during our weekly meetings throughout the year. I would also like to thank John Stell for his useful feedbacks on the mid-project report, allowing research and reflection to be pushed further. Finally, I would like to thank Sarah for her proofreading: she enlightened some grammar mistakes in English that I will not make any more.
# Contents

## 1 Introduction
- 1.1 Overview ......................................................... 1
- 1.2 Aim ............................................................... 1
- 1.3 Objectives ....................................................... 2
- 1.4 Minimum requirements ......................................... 2
- 1.5 Deliverables ..................................................... 2
- 1.6 Schedule ......................................................... 3

## 2 Background research
- 2.1 Background on finite automata ................................. 5
  - 2.1.1 Alphabet, strings and languages .......................... 5
  - 2.1.2 Deterministic finite automata ............................ 6
  - 2.1.3 Nondeterministic finite automata ....................... 7
  - 2.1.4 Determinisation: turning an NFA into a DFA ........... 8
- 2.2 Minimisation of DFAs ........................................... 9
  - 2.2.1 Hopcroft’s algorithm ....................................... 10
  - 2.2.2 Brzozowski’s algorithm .................................... 11
  - 2.2.3 Minimum DFA to test automata equivalence .......... 12
- 2.3 Reducing the size of NFAs ..................................... 13
  - 2.3.1 Enumeration of NFAs ....................................... 13
  - 2.3.2 Reverse automaton method ............................... 15
  - 2.3.3 Equivalence of states in NFAs ......................... 16

## 3 Tools choices
- 3.1 A similar software: JFLAP .................................... 19
3.2 Programming language: Java .................................................. 20
3.3 Input and output format: DOT language ................................. 20
3.4 GUI: Swing library (Java) ...................................................... 22

4 Development ................................................................. 24
4.1 Representation of automata and simple operations .................... 24
4.2 Algorithms implementation .................................................. 25
  4.2.1 Determinisation ......................................................... 26
  4.2.2 DFA minimisation ..................................................... 26
  4.2.3 Test of NFA equivalence .............................................. 27
  4.2.4 Accessible and co-accessible states ................................. 27
  4.2.5 Reversal and reverse method for NFAs ............................ 28
  4.2.6 Complement ............................................................ 29
  4.2.7 NFAs enumeration .................................................... 30
  4.2.8 States equivalence .................................................... 32
4.3 Software–user interactions ................................................ 32
  4.3.1 File I/O ................................................................. 33
  4.3.2 Parsing DOT files ...................................................... 33
  4.3.3 Graphical user interface .............................................. 35

5 Evaluation ................................................................. 37
5.1 General evaluation of the project ...................................... 37
5.2 Randomly generated automata .......................................... 39
  5.2.1 Idea ................................................................. 39
  5.2.2 Creation of random automata ..................................... 39
  5.2.3 Results ............................................................. 39
  5.2.4 Conclusion and limits of randomly generated automata .......... 42

6 Conclusion ................................................................. 43
6.1 Meeting objectives ......................................................... 43
6.2 Further work ............................................................. 44

A Personal reflection ....................................................... 48

B External material .......................................................... 50
C Ethical issues
D Requirements and advice to use the software
Chapter 1

Introduction

1.1 Overview

Regular expressions are an algebraic way of defining languages [12]. They describe how the words of a language can be generated from simpler elements, combining them thanks to a range of operations. The languages that can be described by regular expressions are in fact exactly the same as those defined by devices called finite automata, abstract machines that recognize if a given string belongs to a language or not. Regular expressions allowing a more natural and declarative way to express the words recognized, they are the basis of many applications handling strings, for example text search or lexical analysis. This project will not deal with regular expressions, but the subject can be explored further by reading Chapter 3 of [12] or Section 3.1 of [17]. However, they can be easily converted into finite automata (nondeterministic ones in particular) to store their representation in memory. Thus automata are widely used in area such as programming languages, editors, compilers, parser [8, 14] or even biology (genetics and neuronal system) [7]. Looking for smaller NFAs is a useful task, because their size is naturally linked to the running time of the algorithms based on them.

1.2 Aim

The aim of this project is to find and to implement methods that reduce the number of states of nondeterministic finite automata (NFA) in some special cases. The minimisation of NFAs is an open problem, so I do not attempt to achieve this for any input, or always to get the smallest possible NFA.
1.3 Objectives

The objective of this project is to create a piece of software that applies the techniques found to reduce the number of states of nondeterministic finite automata, that is:

- it receives an NFA $A$ in input;
- it finds the minimum DFA (deterministic finite automaton) equivalent to $A$. If this minimum automaton has fewer states than $A$, this gives already an upper bound for the number of states of a reduced NFA;
- it uses other techniques to try to find a NFA equivalent to $A$ with fewer states;
- if no smaller automaton is found, it says it cannot do anything better than $A$.

1.4 Minimum requirements

The system developed should meet these minimum requirements:

- convert an NFA into an equivalent DFA
- given a DFA, find the minimum equivalent DFA
- check whether two NFAs are equivalent
- given a DFA accepting $L$, construct an NFA accepting the reversal of $L$
- given a DFA accepting $L$, construct an NFA accepting the complement of $L$

1.5 Deliverables

The deliverables will consist of:

- the final report, which will present the different ways I have found to reduce the number of states of an NFA;
- the software system implementing these methods where possible.
1.6 Schedule

The Gantt chart on the next page shows my plan for the project achievement. The details for the implementation of operations to reduce NFAs are:

- enumeration of NFAs (see Section 2.3.1);
- reverse method (see Section 2.3.2);
- equivalence of states (see Section 2.3.3);
- other ideas, if any.
Chapter 2

Background research

I present here the reading and research I have done to undertake this project. The background on finite automata is mainly from three textbooks [12, 3, 17]. The methods to reduce NFAs are either classic ideas or from an article by Ilie and Yu [13].

2.1 Background on finite automata

It is necessary to have a good knowledge of finite automata before tackling the main problem of this project. In this section I will present all the basic notions.

2.1.1 Alphabet, strings and languages

The concept of language is at the center of automata theory, because finite automata are in fact designed to describe some languages [12].

An alphabet is a finite non-empty set, whose elements are called symbols (or letters). It is often denoted by $\Sigma$.

A string (or word) over an alphabet is a finite sequence of symbols from this alphabet. For example, 2201 is a string over the alphabet $\Sigma = \{0, 1, 2\}$. The length of a string is the number of symbols in it. For example, the length of the string 2201, denoted by $|2201|$, is 4. There is a unique string of length 0, called the empty string, and denoted $\varepsilon$. We define the power of an alphabet, $\Sigma^k$, to be the set of strings of length $k$ with symbols in $\Sigma$. Note that $\Sigma^0 = \{\varepsilon\}$ for any alphabet $\Sigma$. $\Sigma^* = \bigcup_{k \in \mathbb{N}} \Sigma^k$ is the set of all finite strings of $\Sigma$.

A language over an alphabet $\Sigma$ is a set of strings in $\Sigma^*$ (in other words, a subset of $\Sigma^*$). For example, the sets $L_1 = \{\varepsilon\}$, $L_2 = \{111, 220, 1, 01\}$ and $L_3 = \Sigma^6$ are languages over the alphabet
\[ \Sigma = \{0, 1, 2\} . \]

### 2.1.2 Deterministic finite automata

I will first introduce deterministic finite automata (or DFA), because they correspond more naturally to the idea of how a machine works. A DFA has got a set of states, and at any time it is in current state. Then, depending on an input symbol, it will swap for another state.

In a more formal definition [3], a deterministic finite automaton (DFA) is a 5-tuple \( A = (Q, \Sigma, \delta, q_0, F) \) where:

- \( Q \) is a (non-empty) finite set of states;
- \( \Sigma \) is an alphabet, containing the input symbols;
- \( \delta \) is a transition function, \( \delta : Q \times \Sigma \rightarrow Q \);
- \( q_0 \in Q \) is the initial (or start) state;
- \( F \subset Q \) is the set of final (or accepting) states.

The transition function describes how to change from one state to another, with a given symbol. But it is also possible to extend this function from single symbols to strings, where there are successive inputs corresponding to the sequence of symbols in the string. So the extended transition function [3] for \( A \) is the function \( \hat{\delta} : Q \times \Sigma^* \rightarrow Q \) defined recursively as follows:

1. \( (\forall q \in Q) \quad \hat{\delta}(q, \varepsilon) = q \)
2. \( (\forall q \in Q)(\forall a \in \Sigma) \quad \hat{\delta}(q, a) = \delta(q, a) \)
3. \( (\forall q \in Q)(\forall a \in \Sigma)(\forall w \in \Sigma^*) \quad \hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w) \)

A accepts a string \( w \in \Sigma^* \) if \( \hat{\delta}(q_0, w) \in F \). In other words, the path labelled by \( w \) from \( q_0 \) leads to a final state. In the other case, \( A \) rejects \( w \). Then, the language accepted by a DFA is the set of all accepted strings, that is \( L = \{ w \in \Sigma^* : \hat{\delta}(q_0, w) \in F \} \). The languages that can be accepted by a DFA are called the regular languages.

Finite state automata are usually represented by a graph called transition (or state) diagram [12]: the states are nodes, and the transitions are represented by arcs from a state to another with the corresponding symbol. For example, let us consider the finite automaton
A = ({q₀, q₁}, {0, 1}, δ, q₀, {q₁}) with transitions q₀ →₀ q₀, q₀ →₁ q₁, q₁ →₀ q₁ and q₁ →₁ q₀. The language accepted by A is the set of strings over the alphabet \{0, 1\} containing an odd number of 1. This DFA is represented by the following transition diagram:

![DFA Transition Diagram]

2.1.3 Nondeterministic finite automata

This is the type of automaton I am particularly interested in for this project. It is nondeterministic in the sense that, for a given state and input symbol, it is possible to have several next states, or not at all.

A nondeterministic finite automaton (NFA) [3] is a 5-tuple \(A = (Q, \Sigma, \delta, q_0, F)\) where \(Q\), \(\Sigma\), \(q_0\) and \(F\) are the same as for DFAs, but the difference is in the type of \(\delta\). In NFAs, the transition function maps a single state and a symbol to a set of states, whereas it is exactly one state in DFAs. So here \(\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)\) (where \(\mathcal{P}(Q)\) is the set of subsets of \(Q\)). A DFA is a particular NFA, where for a fixed state and symbol, the set of states obtained by \(\delta\) contains exactly one state. Sometimes, NFAs are defined with a set of initial states, but I chose to consider only one initial state for simplicity.

As for DFAs, the extended transition function for NFAs can be defined. This is a function \(\hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)\) defined recursively as:

1. \((\forall q \in Q)\) \(\hat{\delta}(q, \varepsilon) = \{q\}\)

2. \((\forall q \in Q)(\forall a \in \Sigma)(\forall w \in \Sigma^*)\) \(\hat{\delta}(q, wa) = \bigcup_{p \in \hat{\delta}(q, w)} \delta(p, a)\)

Then, we say that A accepts a string \(w \in \Sigma^*\) if \(\hat{\delta}(q_0, w) \cap F \neq \emptyset\). So the language accepted by a NFA is the set of strings such that there exists a path leading from \(q_0\) to a final state. NFAs are usually also represented by transition diagrams.

Although NFAs have the ability to be in several states at once, they are not more powerful than DFAs (see 2.1.4). This means that the set of languages that can be accepted by an NFA is the same as for DFAs: the regular languages. Then, one could wonder what is the point of dealing with NFAs: it seems simpler to have a unique transition per symbol. But NFAs are usually easier to construct for a given language. And above all, for a given language \(L\), the
minimum DFA accepting $L$ can be up to exponentially larger than an NFA accepting $L$: this is the case in particular for the languages $L_k = \{u1v : |v| = k\}$ over the alphabet $\{0, 1\}$.

### 2.1.4 Determinisation: turning an NFA into a DFA

Two automata are said to be *equivalent* if they accept the same language. As expressed earlier, for every NFA there exists a DFA accepting the same language, in other words an equivalent DFA. Such a DFA can be found by a construction called the *subset construction* (or *powerset construction*)[12].

Given an NFA $A = (Q, \Sigma, \delta, q_0, F)$ the algorithm consists in constructing a DFA whose states are the subsets of $Q$. This DFA is explicitly defined as $A_D = (\mathcal{P}(Q), \Sigma, \delta_D, q_0, F_D)$ where $\mathcal{P}(Q)$ is the set of subsets (or power set) of $Q$, $F_D = \{X \in \mathcal{P}(Q) : X \cap F \neq \emptyset\}$ (a state $X$ is final if it contains a final state of $A$), and $\delta_D : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$ is defined by:

$$\delta_D(X, a) = \bigcup_{q \in X} \delta(q, a)$$

That is, there will be a transition in $A_D$ from state $X$ to state $Y$ if and only if there exists a transition in $A$ from a state in $X$ to a state in $Y$. This new automaton is a DFA by our definition.

As the set of states of $A_D$ is the powerset of $Q$, it contains $2^{|Q|}$ states. However, there are often some inaccessible states, that we can get rid of while constructing this automaton. In a finite automaton, a state is called *accessible* if there is a string leading from the start state to this state. An automaton is called *connected* if all its states are accessible. To construct a DFA with the subset construction, we can therefore begin from the start state and, following the transitions, create only the accessible states. For example, consider the NFA represented below. It accepts all the strings over alphabet $\{0, 1\}$ whose last or second last symbol is 0.

This automaton is clearly not a DFA: from state $q_0$ with symbol 0 we can either stay on $q_0$ or go to $q_1$, and from state $q_2$ there are no possible transitions for any symbol. We want to construct a DFA equivalent to this NFA. The start state of the DFA will be $\{q_0\}$. From this state, with
input symbol 0, we go to a state corresponding to the union of the states in the NFA, that is, \( \{q_0, q_1\} \). With 1, we stay on \( \{q_0\} \). From \( \{q_0, q_1\} \), 0 brings to \( \{q_0, q_1, q_2\} \), because we have the transitions \( q_0 \xrightarrow{0} q_0, q_0 \xrightarrow{0} q_1 \) and \( q_1 \xrightarrow{0} q_2 \) in the NFA, and 1 brings to \( \{q_0, q_2\} \) because of \( q_0 \xrightarrow{1} q_0 \) and \( q_1 \xrightarrow{1} q_2 \). We go on with this process, and we stop when all the accessible states have been created. We obtain the following transition table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>{q_0}</td>
<td>{q_0, q_1}</td>
<td>{q_0}</td>
</tr>
<tr>
<td>{q_0, q_1}</td>
<td>{q_0, q_1, q_2}</td>
<td>{q_0, q_2}</td>
</tr>
<tr>
<td>{q_0, q_2}</td>
<td>{q_0, q_1}</td>
<td>{q_0}</td>
</tr>
<tr>
<td>{q_0, q_1, q_2}</td>
<td>{q_0, q_1, q_2}</td>
<td>{q_0, q_2}</td>
</tr>
</tbody>
</table>

The construction of the DFA is now easy, just notice that the final states are the states containing one of the final states of the NFA (which were \( q_1 \) and \( q_2 \)): \( \{q_0, q_1\} \), \( \{q_0, q_2\} \) and \( \{q_0, q_1, q_2\} \). The equivalent DFA is represented here:

For this automaton, not all the \( 2^3 = 8 \) states were constructed, but it has still more states than the NFA.

### 2.2 Minimisation of DFAs

Working with small finite automata would be easier. Although finding small NFAs is not an easy task (as it will be seen in the next section), there exist algorithms to minimise DFAs. And more than just reducing the number of states, they will give a DFA with the fewer possible number of states. This minimum DFA is essentially unique: given two minimum DFAs that are equivalent, there is a way to rename the states so that the DFAs become the same.
2.2.1 Hopcroft’s algorithm

The two main algorithms to minimise DFAs are Hopcroft’s and Moore’s. These algorithms are based on the notion of equivalent states. Two states \( p \) and \( q \) of a DFA are said to be equivalent if for all string \( w \), \( \hat{\delta}(q, w) \in F \) if and only if \( \hat{\delta}(p, w) \in F \) [12]. In other words, starting from the states \( p \) and \( q \), there is no input string that can distinguish them: the string will either lead to a final state in both cases, or to a non-final state in both cases. In a DFA, if such equivalent states are merged, the DFA obtained is equivalent. Moreover, the automaton obtained by merging all equivalent states of a DFA is the minimum DFA (this is demonstrated in Section 4.4.4 of [12]).

Here I will explain in details Hopcroft’s algorithm, described by Hopcroft, Motwani and Ullman in [12]. It consists in finding all the pairs of distinguishable states, which means that there is a string \( w \) that distinguishes them, and the remaining pairs correspond to equivalent states. For a given DFA \( A = (Q, \Sigma, \delta, q_0, F) \):

- if \( p \in F \) and \( q \in Q \setminus F \) then the pair \( \{p, q\} \) is distinguishable: indeed the empty string \( \varepsilon \) distinguishes them;

- if \( p, q \in Q \), and \( a \in \Sigma \) is an input symbol such that \( s = \delta(p, a) \) and \( t = \delta(q, a) \) are known distinguishable, then \( \{p, q\} \) is a pair of distinguishable states: indeed, for \( w \in \Sigma^* \) that distinguishes \( s \) and \( t \), \( aw \) distinguishes \( p \) and \( q \).

Let us take a simple example to understand this process. Consider the automaton obtained after it was turned into a DFA in the previous section, and rename its states:

![Automaton Diagram]

We are going to fill in the table of state inequivalences for this DFA. \( \{q_0, q_1\}, \{q_0, q_2\} \) and \( \{q_0, q_3\} \) are distinguishable pairs, because \( q_1, q_2 \) and \( q_3 \) are final states but \( q_0 \) is not. The pair \( \{q_1, q_2\} \) is then distinguishable because \( \delta(q_1, 1) = q_2 \) and \( \delta(q_2, 1) = q_0 \) are distinguishable. Likewise, \( \{q_2, q_3\} \) is distinguishable. Then the process ends: \( q_1 \) and \( q_3 \) are not distinguishable because for any symbol, their transitions go to the same state. The minimum DFA for this...
example is thus obtained by merging states $q_1$ and $q_3$. The table constructed and the minimum DFA are shown below.

\[
\begin{array}{c|ccc}
q_0 & x & x & x \\
q_1 & x & x & x \\
q_2 & & & \\
q_3 & & & \\
\end{array}
\]

\[
\begin{array}{c|cc}
p_0 & 0 & 1 \\
p_1 & 1 & 0 \\
p_2 & & \\
p_3 & & \\
\end{array}
\]

2.2.2 Brzozowski’s algorithm

Another algorithm for DFA minimisation should be presented, namely Brzozowski’s algorithm. Unlike the algorithms given by Hopcroft or Moore, which are based on the idea of merging equivalent states, this one only uses determinisation (seen in Section 2.1.4) and the reverse of an automaton that I shall define now.

Reverse of an automaton

The reverse (or reversal) of a string $w = a_1a_2...a_n$, where the $a_i$ are symbols, is $w^R = a_n a_{n-1}...a_1$. It is then also possible to define the reverse of a language $L$ as $L^R = \{w^R : w \in L\}$. From an NFA accepting a language $L$, it is possible to construct very easily an automaton accepting $L^R$. And this construction is going to help us getting a smaller NFA in some cases.

Given an NFA $A$ that accepts a language $L$, we want to construct the NFA $A^R$ that accepts $L^R$. The principle is based on simple operations applied to $A$:

- Reverse all the transitions in $A$, that is, for every transition from a state $p$ to a state $q$ labelled by a symbol $a$, we replace it by a transition from $q$ to $p$ with symbol $a$;

- Exchange the initial and final states. A problem here is that if $A$ has more than one final state, we get an automaton with several initial states, and this does not correspond to our definition of an NFA. This can be resolved by adding one state $q$ that will be the only initial state, and for each transition from one of the several initial states, we add a transition with the same symbol to the same state, but from $q$. We also set $q$ final if one of the initial states was final.

Note that this process can create inaccessible states that can be removed. The resulting NFA is the automaton $A^R$ we were looking for. The following example shows an NFA accepting the
strings over \(\{0,1\}\) starting with '1' or '01' (on the left), and its reverse accepting the strings ending with '1' or '10' (on the right):

**Brzozowski’s construction**

Brzozowski’s algorithm does not appear to be as widely known as Hopcroft’s and Moore’s algorithms since it is not mentioned in the three textbooks on automata theory that were mainly used for this project [12, 3, 17]. This algorithm is presented by Brzozowski in [2], but Tabakov and Vardi [18] describe it more formally and quickly, and compare its performance with Hopcroft’s algorithm.

The observation Brzozowski made was that computing the reverse of a DFA \(D\) would give the NFA \(D^R\) which, once turned into a DFA using the subset construction, is exactly the minimum DFA for \(D^R\). His algorithm consists then in applying twice the reverse followed by a determinisation, so as to get the minimum DFA for the original automaton. So from a finite automaton \(A\) (possibly nondeterministic), the process is:

- compute the reverse \(A^R\) of \(A\);
- turn \(A^R\) into a DFA \(B\);
- find the reverse \(B^R\) of \(B\);
- determinise \(B^R\) to get the DFA \(C\), which is the minimum DFA of \(A\).

**2.2.3 Minimum DFA to test automata equivalence**

Recall that two automata are equivalent if they recognize the same language. Since for a given regular language there exists a unique minimum DFA, a way to check whether two NFAs are equivalent is to compute their minimum DFA, and to see if they are the same (this may need to rename the states). This notion of equivalent automata is at the heart of this project because
the goal is to construct automata that are smaller while remaining equivalent to a particular automaton.

2.3 Reducing the size of NFAs

It would be very convenient to find as well a minimum NFA for a regular language \( L \), since it could have fewer states than the minimum DFA. But this is not that simple. Firstly, an NFA with the minimal number of states for a given language would not be unique, and an example is given with the following automata. Those NFAs both accept the language of all the strings over \{0, 1\} starting with a '0'. And it is not possible to construct an equivalent NFA with only one state: the initial state of such an automaton should not be initial as the empty string is rejected, but the language accepted is not empty so there should be at least one accepting state, which is a contradiction.

Then, the method of merging equivalent states used earlier for DFAs will not produce a minimal NFA (see Section 2.3.3).

A first step to reduce an NFA could be to create the minimum DFA accepting the same language: it could already be smaller than our NFA, and in this case it is already an upper bound for the number of states of a smaller NFA. But it can also have the same number of states (as in our example), or even more. That is why other techniques need to be found.

2.3.1 Enumeration of NFAs

An idea to find a smaller NFA than a given NFA \( A \) is to enumerate all NFAs with fewer states than \( A \), and check whether they are equivalent to \( A \). Clearly, this is not a very efficient solution, because the number of NFAs to generate will often be too large to make it possible. However, it is possible to apply this method to generate not all the NFAs, but only those with fewer states than a fixed (and small) number.

Let \( A \) be an NFA over alphabet \( \Sigma \) with \( m \) states. Starting with \( n = 1 \), the idea of the algorithm would be to generate all NFAs over \( \Sigma \) with exactly \( n \) states and, for each of them, test the equivalence with \( A \):
• if one of these automata is equivalent to \(A\), then an NFA smaller than \(A\) and accepting
  the same language would have been found;

• if none of them is equivalent, the process starts again for \(n + 1\).

The algorithm stops when it reaches \(m\).

For a fixed number of states, the NFAs are generated with an increasing number of transitions. This way, if an NFA is found equivalent to \(A\), the number of transitions would be minimised as well.

If \(|\Sigma| = k\), let us calculate the number of NFAs with \(n\) states. Let \(Q\) be the set of states for these NFAs (so we have \(|Q| = n\)). Among these states, we have \(n\) possibilities for the initial state, and \(2^n\) for the set of final states. If we consider a transition as a triple \((q_i, s_l, q_j)\) where \(q_i, q_j \in Q\) and \(s_l \in \Sigma\) (with \(0 \leq i, j \leq n\) and \(0 \leq l \leq k\)), the set of possible transitions in the automaton is \(Q \times \Sigma \times Q\), so there are \(|Q \times \Sigma \times Q| = n^2 \times k\) possible transitions. Since we can choose any subset of those transitions to construct an NFA, we have to consider the number of such subsets: \(2^{n^2 \times k}\). The total number of automata we would thus have to enumerate is

\[
n \times 2^n \times 2^{n^2 \times k} = n \times 2^{n^2 \times k+n}
\]

The number of automata to generate is exponentially larger than the number of symbols in the alphabet and states. So it will be impossible to apply this to a large alphabet or number of states. However this number can be reduced, considering the fact that NFAs generated by this method will be isomorphic or equivalent to many others; we can find ideas to avoid constructing all those NFAs, and thus improve the algorithm:

• Instead of choosing the initial state among \(n\), we can simply fix it. Indeed, for \(p, q \in Q\),
each NFA with initial state \(p\) is equivalent to an NFA with initial state \(q\) by renaming
the states (\(p\) becomes \(q\) and \(q\) becomes \(p\)). So the number of generated automata can be
  divided by \(n\).

• The set of final states is not very important either. The only parameters that matter are
  the number of final states, and if the initial state is final or not (and by renaming the
  states, we can then get all possible combinations). Thus instead of \(2^n\), the number of
  choices for final states is reduced to \(2 \times n\), where 2 corresponds to choosing if the initial
  state is or not final, and \(n\) to the choice of the number of final states among the \(n - 1\)
  remaining states (from \(0\) to \(n - 1\)). The initial state being initial if and only if the empty
string is accepted, a prior test on the original automaton could even divide this number by 2.

- If \( n \geq 2 \), we can ignore the case where an NFA has no final states, because this means that the language accepted is the empty set, and thus this NFA is equivalent to an automaton with only one state, which should have been constructed earlier if we follow the algorithm described above.

- Each automaton containing inaccessible states is equivalent to an NFA with fewer states, so we should try to avoid constructing them. To this end, a simple idea is to start with at least \( n - 1 \) transitions for NFAs with \( n \) states. Other ideas can be found to avoid disconnected automata.

Although this algorithm is not very efficient, if we find an equivalent automaton with this method we are sure that it is one of the minimum NFAs.

### 2.3.2 Reverse automaton method

Here is a method using the reverse of a finite automaton described in Section 2.2.2. A notable point of the reverse construction is that the reverse of the reverse of an automaton recognizes the same language as itself. This property was already used in Brzozowski’s algorithm, and this is with a very similar idea that the method called reverse automaton method works.

This method applies successively four other methods: reverse, minimum DFA, reverse again and removing inaccessible states. Let us explain how this reverse construction can be used to reduce the number of states in NFAs. A fact to remember is that it has been shown in Section 2.2.2 that the reverse had at most one more state than the original automaton. Given a regular language \( L \), we can construct the minimum DFAs \( A_1 \) accepting \( L \) and \( A_2 \) accepting \( L^R \) (note that the reverse of a regular language is also regular). Let \( m \) be the number of states in \( A_1 \) and \( n \) the number of states in \( A_2 \). It is possible that one of these DFAs is smaller than the other one, say \( A_2 \). If we have moreover \( n < m - 1 \), then \( n + 1 < m \), and as \( A_2^R \) (constructed as explained before) has at most \( n + 1 \) states, this is an NFA accepting \( L \) that has fewer states than the minimum DFA \( A_1 \). But even if the condition \( n < m - 1 \) is not fulfilled, \( A_2^R \) can still be smaller than \( A_1 \) after deleting its inaccessible states. So if we had at the beginning an NFA \( A_0 \) accepting \( L \), we would get a smaller automaton by minimising the reverse than by minimising \( A_0 \) itself.
This method works better for some languages than others, in particular the languages $L_k = \{u1v : |v| = k\}$ over the alphabet $\{0, 1\}$. For example, consider the NFA represented by this diagram:

![NFA Diagram](image)

The language accepted is $L_2$, but let us pretend we did not notice, and see how the reverse method works on it. If we turn this NFA into a DFA, we will obtain an automaton with 12 states. And even after DFA minimisation, the automaton still have 8 states. But if we first take the reverse of this automaton, then find the minimum DFA and reverse it again, the NFA obtained (after removing an inaccessible state) has only 4 states, which is fewer than the starting NFA (see this automaton below). We succeeded in constructing a smaller NFA.

![Reduced NFA Diagram](image)

Although this worked well on this particular example, this is not always the case.

The method described here is in fact close to Brzozowski’s algorithm to find the minimum DFA, because it uses a double reverse. The differences are that instead of determinise the reverse, we minimise it, and in the end we avoid the extra determinisation because what we are looking for is an NFA. I was unable to find an exact description of this method with the particular aim to reduce NFAs in any textbook or article.

### 2.3.3 Equivalence of states in NFAs

Another way to reduce the number of states in NFAs would be to find an equivalence relation over the states, such that we can merge equivalent states without changing the language accepted by an automaton. This idea is in fact very similar to DFA minimisation, but for NFAs it is
more complicated to find a suitable equivalence relation. Ilie and Yu [13] give a good solution for this problem.

As we did earlier for DFAs, we are going to look for indistinguishable pairs of states, that could thus be merged. In an NFA, two states $p$ and $q$ are distinguishable if there is a string $w$ that can lead from $p$ to a final state, but cannot lead from $q$ to a final state. With nondeterminism, we have no longer one state, but a set of states for a single transition, so it is more complicated to decide whether two states are distinguishable or not.

Finding all the states that could be merged in NFAs would be too expensive. So we are going to consider an equivalence relation, denoted by $\equiv_R$, such that any two equivalent states are indistinguishable, but non-equivalent states are not necessarily distinguishable. In other words, this will enable us to merge some states without changing the language, but our relation will not find every possible merging.

To compute this equivalence relation, as for DFAs, we look for the states that are not equivalent. The definition of the non-equivalence $\not\equiv_R$ in an NFA $A = (Q, \Sigma, \delta, q_0, F)$ can be expressed recursively as follows:

1. non-final states are not equivalent to final states;

2. two states $p$ and $q$ are not equivalent if, for a symbol $a$, there is a state in $\delta(p, a)$ that is not equivalent to any state in $\delta(q, a)$.

Note that $\delta(p, a)$ and $\delta(q, a)$ are not empty, since we will always complete the considered NFA with an extra state where all the non-existing transitions will lead.

An algorithm to compute non-equivalent states of $A$ is:

- add a new state $x$ to $Q$, non-initial and non-final, and for every state $q$ and symbol $a$ such that $\delta(q, a) = \emptyset$, set $\delta(q, a) = \{x\}$. Also, for every symbol in $\Sigma$, add a transition from $x$ to itself;

- $\forall p \in Q \setminus F \forall q \in F$ ($\{p, q\}$ is a non-equivalent pair);

- while there are $p, q \in Q$ such that, for some symbol $a$, there exists a state in $\delta(p, a)$ not equivalent to any state in $\delta(q, a)$, add $\{p, q\}$ to the set of non-equivalent pairs, and continue.

Then the equivalent states correspond to all pairs of states that have not been marked as non-equivalent. The NFA obtained by merging these equivalent states (and removing the state $x$
we created) is equivalent to $A$, and it is smaller. This technique not only reduces the number of states, but also the number of transitions in an NFA. But the limitations are that:

- Not all the indistinguishable pairs are found, so there could exist some more merging possibilities. For instance, in the following example, we could merge states 1 and 2 of the automaton on the left and get the equivalent NFA on the right, but the algorithm described above will not find this possibility:

- Sometimes, some states of an NFA should be split in order to find ways to merge them differently. Coulon [7] gives the following example, where state 1 of the automaton on the left has to be split before it is merged either to state 2 or state 3, to get the minimum NFA on the right:
Chapter 3

Tools choices

In preparation of the implementation, several choices were made concerning the tools to be used for the development of the software system. This implementation task consisting mostly in algorithms implementation (see Section 4.2), no specific system development methodology was followed: correctness of those algorithms was the main concern here.

3.1 A similar software: JFLAP

A first approach to understand what a software system working on finite automata could do was try to find similar programs. The research of smaller NFAs is a rare subject and although several papers exist in that area, no software with this particular aim was found. However, a few systems exist dealing with finite automata in general. JFLAP is one of them [16]: it is intended for students, allowing them to work with finite automata, as well as other structures such as Turing machines, grammars or regular expressions. The principle of this system is creating automata, and then acting on them with a certain number of provided operations. The examination of JFLAP was a good starting point to imagine what a system about finite automata could do. But the objective was not to copy JFLAP or to produce something similar because:

- First my own software had to deal specifically with the reduction of NFAs, and that approach was not tackled in JFLAP;

- Then the target audience was not the same: JFLAP is directed to students, with a user-friendly environment and an educational goal, whereas I wanted my system to be able to help in real applications of finite automata, so the emphasis was not on the interface.
Thus I made the decision to create a new software system, no part or source code from JFLAP was used. This way, it is possible to build the model of an automaton with the interesting characteristics for the pursued objective only, avoiding features that would have been useless for this project. Besides, I think that exploring the code of JFLAP to understand its mechanisms would have been too long. Starting a new program seemed in all aspects a much better idea.

JFLAP raised the question of the input of an automaton. In this system, the input is made either by mouse clicks to create new states and link them with transitions, or by a file in XML format. For my software, a different option was chosen, as explained in Section 3.3.

3.2 Programming language: Java

The chosen programming language for this system was Java. There are two main reasons for this choice. First, Java is an object-oriented programming language, and the idea of considering states and automata as objects seemed very convenient. This is also a language I have a good prior knowledge of, because I learnt it at university and I have already used it for several other projects. However, the a book by Harrington [11] was of good help during the implementation, when I could not remember something.

A Java object has a certain amount of attributes represented by variables, and methods that can operate on it. Two main types of objects were thus defined: State and Automaton. From this, we could implement methods acting directly on them (implementation is detailed in Chapter 4). Furthermore, Java provides a wide choice of libraries, making the programming much more comfortable [15]. In particular, different classes implementing the Collection interface were used, to represent collections of objects.

3.3 Input and output format: DOT language

After considering the programming itself, the way that the system would interact with the user had to be decided. A first question was to know how would a user specify an automaton to the system. JFLAP for example can take input file in XML format, but the automaton can also be created directly thanks to the graphical user interface, by “drawing” the transition diagram with the mouse. This last method clearly being too much work, and being anyway not really necessary for the goal of this project, file input appeared as the natural option. In spite of the fact that XML is a standard format in many applications, it was not selected: DOT language
was chosen for this task.

DOT is a file format used to describe graphs, either directed or not. It allows us to choose the number of nodes and to dispose edges between them, with the possibility to give labels. This format is used by a program called dot, which is part of the GrapViz project, to draw the graph described by such a file. Refer to GraphViz website [9] to know the exact grammar of DOT language. The necessary conditions for a DOT file to be accepted by my system are defined in Section 4.3.2. Here, we will only present an example of a DOT file containing the basic elements to describe a finite automaton:

digraph automaton {
    start [style=invis, shape=point];
    start->q0;
    q4 [peripheries=2];
    q0->q0 [label="1"];
    q0->q1 [label="1"];
    q0->q2 [label="0"];
    q1->q3 [label="1"];
    q1->q3 [label="0"];
    q2->q1 [label="1"];
    q2->q2 [label="1"];
    q2->q2 [label="0"];
    q3->q4 [label="1"];
    q3->q4 [label="0"];
    q5;
}

This file describes a directed graph called `automaton`. Each line ending with “;” between the braces is an instruction. The first two instructions define `q0` as the initial state: the first one defines an invisible point called `start` and the second one defines an arrow from this point to state `q0`. The third instruction is the notification that `q4` is a final state (it will have two periphery lines). All following instructions (except for the last one) describe the transitions: before and after the arrows `->` are the name of the states, and the `label` is the symbol of the transition. The last instruction defines a state with name `q5`, but it is not necessary to define all the states this way, because if their name appears in an instruction of initial state, final state or transition, they are automatically defined. For example in this file, six states have in fact been defined: `q0`, `q1`, `q2`, `q3`, `q4` and `q5`.

Since the software developed for this project aimed to work on NFAs, dot seemed the perfect tool to create pictures of considered automata, in order to include them in the graphical user interface. From there, the decision was made to use this same format to store an automaton in
memory with a view to operate on it. This way, a single description of an automaton in DOT language could be used by two applications: dot, and my own software system. The calling of dot executable from my system is made with the help of an API called GraphViz Java API [1], that I did not implement myself.

An idea could now be to convert the representation of an automaton from DOT format to XML format to make my system compatible with JFLAP, and this would not be very complicated to implement. But I decided to focus on the most interesting work, that is the implementation of efficient methods to reduce NFAs, instead of considering many side features that would not have always been useful depending on the use made of such a software system.

### 3.4 GUI: Swing library (Java)

The graphical user interface (GUI) of the software system was implemented using Swing GUI, from the Java API. The Swing package provides many useful tools to create a friendly environment for the user: it makes it possible to display windows, to include pictures, to place buttons or menus and to give them specific actions when receiving a mouse click, etc. It has become the standard Java GUI, replacing the old one, AWT [6]. Thus choosing Swing is probably the best option when programming in Java.

I had a previous quick try at graphical programming with Swing, but this was still a bit long to remember or learn again its operational principle. The construction of a GUI starts with the creation of a first container which will gather all the features. It can contain other containers, or components, that is items like buttons, menus or text. Those components are organised in their container by a layout manager which decides of their location in this container. For this GUI to do more than simply displaying things, objects called listeners can define consequences of an external action, for instance pushing a button. This way, the program will be able to “listen” to the user, and act accordingly. A screenshot of what have been achieved for this project is displayed below, and explanations on how this was developed will be given in Section 4.3.3. For further information on Swing, see Chapter 9 of [6].
Figure 3.1: Screenshot of the software interface
Chapter 4

Development

The theoretical research gave some ideas to follow in order to implement algorithms capable of reducing the size of some NFAs. Now, the development phase has to apply those ideas: a Java program shall be built to carry out those methods. The end of this development is the creation of an executable JAR file.

4.1 Representation of automata and simple operations

An automaton is characterised by a set of states, an initial state and an alphabet. Each state can be either initial or not, final or not, has a unique identification number, and has a certain number of transitions to other states labelled with symbols of the alphabet. States and automata correspond each to a Java class. Classes provided by Java, `HashSet` and `HashMap`, were used to represent sets of states or symbols (alphabet), and transitions which are the association of a symbol and a set of states that it leads to. An automaton is well defined if, in its set of states, only one is initial.

The transitions of an automaton could have been described by its state transition table, implemented by a 2-dimensional array. But this format has the disadvantage of forcing the alphabet and the set of states to be permanently fixed. On the contrary, with the mapping approach that was chosen, it is possible and easy to add or delete transitions with a new symbol, or states. Here, each state “knows” which are its successors. By default, an automaton is nondeterministic, because for each symbol of the alphabet, a state has a set of successors. But if every such set contains exactly one state, then the automaton is deterministic.

The construction of an automaton is done as follows:
First, the states have to be created, and each of them is given an ID number;

Using the mutator methods implemented, we can set a state initial or final;

Another method enables one to define a transition from a state, by specifying the symbol of the transition and the state it leads to;

After giving all the desired characteristics to the states, an automaton is constructed by being given a set of such states. The states are added to the automaton with their successors, that is if one of them has a transition to a state not in the set, the latter will be added as well: this is a recursive construction. The alphabet is either deduced from the labels of the transitions, or can be specified independently.

Apart from getters and setters, other class instance methods have been implemented, in particular:

- It is possible to test if the automaton is a DFA. This is done by checking if each state has exactly one successor for every symbol in the alphabet;

- The purpose of finite automata being to describe a regular language, by accepting strings that belong to this language and rejecting the others, a method exists to check if a string is accepted by the automaton. In order to do this, we first defined a recursive method \texttt{accept} on states, where a state accepts a string if there exists a path from this state to a final state labelled by this string. Then an automaton accepts a string if its initial state accepts it;

- Since this project is about finding small NFAs, there exists a method comparing the size of two finite automata;

- It will sometimes be necessary to work on an automaton, but without altering the original one, so I implemented a method capable of copying an automaton, by recreating similar states and transitions;

4.2 Algorithms implementation

Automata being now defined, we can deal with the heart of the matter, that is the implementation of algorithms aimed at transforming them.
4.2.1 Determinisation

We have seen in Section 2.1.4 that, from an NFA, the subset construction gives an equivalent DFA. The implementation is made in a way to be as efficient as possible: not all the states corresponding to subsets of the NFA are created, but only those that are accessible (or reachable) from the initial state. This way, some storage space and time are saved.

The method implementing this algorithm associates a set of states of the NFA with a unique state of the DFA. It is recursive and based on depth-first search. It starts from a set \( A = \{q_0\} \) containing as single element the initial state of the NFA, and associates it with the initial state of the DFA \( p_A \). Then, for each symbol \( c \) in the alphabet, it considers the set \( B \) of successors of states in \( A \) (for the moment, only \( q_0 \)) with \( c \). If the set \( B \) has not been encountered yet, a corresponding state \( p_B \) for the DFA is created and it is final if one of the states in \( B \) is final. Otherwise, \( p_B \) is the state of the DFA previously created for \( B \). A transition from \( p_A \) to \( p_B \) labelled by \( c \) is added in the DFA. If state \( p_B \) has just been created, the process starts again from set \( B \) instead of \( A \), before going back to looking at successors of \( A \) with the next symbol in the alphabet. On the contrary, if \( p_B \) existed already, we stop here with set \( B \). Finally, when all the symbols have been considered, the process ends. The automaton obtained is a DFA which is equivalent to the initial NFA.

4.2.2 DFA minimisation

Two algorithms have been described in Section 2.2 to find the minimum DFA corresponding to a given NFA, that is Hopcroft’s and Brzozowski’s algorithms. I chose to implement the minimisation method given by Hopcroft. It would have been wise to use instead the construction of Brzozowski, which is based on other methods that are going to be or have already been implemented (construction of the reverse and determinisation), because this would have saved some code writing. But I became aware of Brzozowski’s algorithm too late, and at that time the implementation was done already. However, it appeared very interesting to have implemented Hopcroft’s method, since a very similar algorithm (described in Section 2.3.3, and implemented as explained in Section 4.2.8) was to be used for merging states in NFAs (so only minor changes will have to be made to implement this new method).

The implementation of this method starts with turning the input into a DFA if it was not the case, and then follows its description given in Section 2.2.1. A 2-dimensional array representing the table of inequivalence is constructed, in which all pairs are initialised to false.
Then, any pair containing a final and a non-final state becomes $\text{true}$, because those states are not equivalent. While non-equivalent pairs are found, the following test is applied to every pair: two states are distinguishable if there is a symbol for which their respective successors are distinguishable. Once all the equivalent states have been found, the merging is made by keeping only the state with the smallest ID among a group of equivalent states. If this group of states contains the initial state, the corresponding state added in the minimum DFA will be the initial state as well.

4.2.3 Test of NFA equivalence

The test of equivalence of NFAs was one of the minimum requirements, since reducing an NFA means in fact finding a smaller equivalent NFA. Two NFAs are equivalent if they accept the same language. An evident way of checking the equivalence of automata is to compute and compare their minimum DFA: if the minimum DFAs are isomorphic (i.e. identical if we rename the states) then the initial automata were equivalent. We know how to compute the minimum DFA from any NFA now, but the comparison of the two resulting DFAs is not that obvious.

From two minimum DFAs $A_1$ and $A_2$, we have to establish a bijection between their states, and check that the same transitions are present in both automata. For them to be isomorphic, it is also necessary that two corresponding states are either both initial or not initial, and either both final or not final. Those tests are done starting with associating the two initial states $p_0$ and $q_0$, and as for the determinisation explained in Section 4.2.1, we then use depth-first search to explore the successive nodes, following the transitions. For each symbol $c$ in the alphabet, we check that the successor $p_1$ of $p_0$ with $c$ is either associated with the successor $q_1$ of $q_0$ with $c$, or has not been allocated a corresponding state yet. In the latter case, and if $p_1$ and $q_1$ are both final or both non-final, we associate them and we start applying to $p_1$ the same test we did for the initial state, before considering the other symbols in the alphabet. But if $p_1$ was already associated with another state than $q_1$, or in the case that one of $p_1$ or $q_1$ is final and the other non-final, $A_1$ and $A_2$ are not isomorphic. When the process ends, and if no such bad cases have been encountered, we conclude that $A_1$ and $A_2$ are isomorphic.

4.2.4 Accessible and co-accessible states

The definition of an accessible state has been given in Section 2.1.4. Similarly, a state of a finite automaton is called $co$-accessible if there exists a path from this state to a final state. We
are interested in states that are inaccessible, or not co-accessible, because they can be deleted from an automaton without changing the language accepted. Indeed, no string can reach an inaccessible state, thus removing it will not change the functioning of the automaton over input strings. Concerning states that are not co-accessible, if we get to such a state for an input string, we know that whatever symbol we will now input, the string will never be accepted by the automaton: the same result would have been obtained if the string had been rejected in the first place, that is if the state was removed. The automaton obtained in this latter case is always an NFA, because after removing a state that was accessible, there are some transitions pointing nowhere.

Two methods have then been implemented:

- The inaccessible states are removed by constructing an automaton from a set containing only the initial state. As explained in Section 4.1, an automaton is created from a given set of states by following recursively the transitions from those states, thus only accessible states are considered;

- The states that are not co-accessible are removed by applying the same method, but to the reverse automaton. Indeed, in an automaton $A$, if the transitions are reversed, and the initial and final states are exchanged, the inaccessible states of the automaton obtained are the non co-accessible states of $A$.

Note that the inaccessible states are naturally absent when we apply our algorithm of determinisation, or in the minimum DFA. On the contrary, some states in those constructions can be non co-accessible. In particular there are sometimes “dead states”, that is states from which all transitions are loops that go back on them. If we removed such states, we would get an equivalent NFA.

### 4.2.5 Reversal and reverse method for NFAs

The reverse or reversal of an automaton $A$ (see Section 2.2.2) is computed by reversing all the transitions, and exchanging the initial and final states. The first step in the implementation is to create a set $\text{rev}$ of states with the same size as $A$, which will constitute the states of the reverse. There is a one-to-one correspondence between these states and the states of $A$. Among the states of $\text{rev}$, those corresponding to final states of $A$ will be added to another set, called $\text{init}$: they are the “initial states” of the reverse, but since in the end there will be only one
initial state, we do not mark them as initial immediately. For each state $p_1$ of $A$, we pick the state $q_1$ in $\text{rev}$ corresponding to $p_1$. If $p_1$ is initial, we set $q_1$ final. If $p_1$ is final, then $q_1$ is added to $\text{init}$. Then, for any transition leading from $p_1$ to another state $p_2$, we add a transition with the same symbol from $q_2$ to $q_1$, where $q_2$ is the state in $\text{rev}$ corresponding to $p_2$. After doing this for every state of $A$, we get something that is almost a finite automata, but a problem remains: $\text{init}$ can contain several states. So three cases should be considered:

- If the set $\text{init}$ is empty, this means that our first automaton $A$ had no final states, that is the language accepted was empty. In this case, we can simply return an automaton accepting all strings: it has a unique state, and transitions with all symbols are loops;

- If $\text{init}$ contains a unique state, this state is the only initial state of the reverse, which can then be created;

- If $\text{init}$ has several states, a new state $q_0$ is created and added to $\text{rev}$ to be the initial state of the reverse. If one of the states in $\text{init}$ is final, then $q_0$ is set final too. And for all transitions from states of $\text{init}$, we add transitions with the same symbols and to the same states from $q_0$. Then, as we only have one initial state, the reverse automaton is well defined.

The reverse automaton method to reduce NFAs, explained in Section 2.3.2, is then very simply implemented. Indeed, it is based on other methods already included in the program: first we construct the reverse automaton as described above, secondly we compute the minimum DFA, then we apply the reverse again and finally we create the connected automaton (that is we remove inaccessible states). This new automaton will not always be smaller than the initial one, but this can happen.

### 4.2.6 Complement

The complement of a language $L$ over alphabet $\Sigma$ is the language $\Sigma^* \setminus L$, that is the language containing all strings over $\Sigma$ except those of $L$. Then, the complement of a DFA $A$ is the automaton where the final and non-final states have been exchanged. Indeed, the strings that were previously accepted will now end on a non-final state, whereas those that were rejected will reach a final state. Thus the implementation of the complement of a DFA is trivial, since it only consists in exchanging final and non-final states.
Apart from being very easy to construct, the complement has an interesting property that we already noticed concerning the reverse: the complement of the complement of DFA $A$ accepts the same language as $A$. Moreover here, it is $A$ exactly. Because of this, I included the construction of the complement in the minimum requirements: I thought that, as for the reverse, we could have applied some changes between two complements, and thus get an equivalent automaton that could possibly be smaller. But a problem of the complement is that it can only be constructed from DFAs, and the result is a DFA as well. Indeed, take the example of this very simple NFA over the alphabet $\Sigma = \{0, 1\}$:

Only two strings are accepted here: 0 and 1. But if we exchange final and non-final states, the new NFA we get:

accepts only the empty string, and not all other strings over the alphabet. This is explained by the fact that NFAs “lack” some transitions, in the sense that a string is sometimes rejected because no transition exists for its next input symbol. Exchanging final and non-final states will not change this fact, and thus the string will still be rejected. On the contrary, this never happens in DFAs.

A double-complement approach will then never give a better result than the minimum DFA. So this method was implemented, but it is actually useless to reduce NFAs.

4.2.7 NFAs enumeration

First the enumeration of NFAs requires to implement several little tools (that will be reused later for the evaluation), the final objective being to implement the algorithm described in Section 2.3.1. These tools are aimed to set an order over all automata, enabling the enumeration from the smallest one to the biggest one, by deciding of the transitions and final states. Because even
for a fixed number of states, we would like to consider first automata with fewer transitions. During the enumeration, the two parameters to know in order to construct an automaton are:

- **The choice of final states.** But this consists actually only in knowing the number of such final states (an integer), and if the initial state is final or not (a boolean): see the explanation in Section 2.3.1. Ordering this choice is easy: we start considering that the initial state is final, and that none of the other states are final. Then, for each iteration, the number of final states is incremented. When this number reaches its maximum, we switch the initial state to non-final, and start again the count of other final states from zero, until the maximum;

- **The transitions to include in this automaton.** For each choice of final states, we have now to enumerate the successive sets of transitions to give to the automaton. This parameter is described by two arrays, both of size \((\text{number of states})^2 \times (\text{number of symbols})\): the first one contains all possible transitions, where a transition is a triple \((\text{state}, \text{symbol}, \text{state})\), and the other one is a boolean array, where each value indicates whether the transition of same index should be included or not. The array containing the transitions is simply filled by enumerating the possible combinations in natural order. The array representing the choice starts with the minimum number of \texttt{true} to be sure the automaton will not always be disconnected (that is, the same number as the number of states). After exploring all possible combinations for this number of transitions, it is incremented, until we reach the maximum (that is all transitions included).

The order being now set, we can create one after another all NFAs for a fixed number of states. Those will be enumerated with increasing number of transitions, so an automaton enumerated after another one will always be bigger. In the main method, an input NFA \(A\) is given as parameter, and what we are looking for is a smaller automaton. We will thus enumerate automata for fewer states than \(A\), starting with one state, then two, etc. During the enumeration process, the idea is to test the equivalence of every NFA created with the input. If an equivalent NFA is found, the method ends and return this automaton, which is one of the smallest existing NFAs equivalent to this input.

In the end, I decided not to include the enumeration of NFAs in the final software system, because this method is not achievable in a reasonable time. For instance, for an alphabet with two symbols, enumerating automata for up to three states only, and testing each time the equivalence with an input, would already take several minutes. And since this execution time
grows exponentially with the number of states, the enumeration for four states is not actually possible. But the idea was still interesting, and this is on a similar idea that the model for generating random NFAs (see Section 5.2) is based.

### 4.2.8 States equivalence

The implementation of the method that merges equivalent states in NFAs, as detailed in Section 2.3.3, is very similar to the implementation I made of the Hopcroft’s minimisation (Section 4.2.2). In the same way, we will construct a table of inequivalence to determine the states to be merged.

The first task here is not to transform the automaton into a DFA, but to make sure that from each state and with any symbol, there exists at least one successor state. This is done by adding an extra state \( x \) (to be removed later), towards which every non-existing transition will point. Additionally, we put transitions from this state to itself with every symbol of the alphabet, in other words this is a dead state. The inequivalence table, which considers all pairs of states including the new state \( x \), is initialised everywhere to `false`, meaning that no pair has been distinguished for the moment. Then the process is similar to the minimisation of DFAs, except that the criterion of non-equivalence is not the same: for every pair \((p, q)\), taking successively every symbol \( c \), we look over the successors of \( p \) and test if one of them is distinguishable from all successors of \( q \) with \( c \). If we find such a successor, then \( p \) and \( q \) are distinguishable, and we indicate this fact by switching them to `true` in the inequivalence table. In the end, the pairs that remained indistinguishable are merged.

### 4.3 Software–user interactions

After dealing with the implementation of algorithms, which was the heart of the development stage, it is still necessary to implement methods enabling a good usability. The choice made, and already detailed in Chapter 3, is to create a graphical user interface, comprising menus and buttons allowing users to work intuitively on NFAs, as well as a means to load files describing automata from the file system. Those features are very useful to demonstrate the capacities of my software system, by providing a good usability. However, if the algorithms implemented were to be used in real applications of NFAs, it would probably be necessary to review the interaction interface so as to allow their inclusion in a bigger system.
4.3.1 File I/O

As decided in Section 3.3, the system developed has to get the input automata from DOT files. It should as well be able to save an automaton in this same format. Java provides several classes to interact with the file system. First, the class File is an abstract representation of a real file, which can be created from the name of this file. But with the GUI (see Section 4.3.3), an easier way to choose files from the system will be introduced. If a file contains something, we can then read it. I chose to use the classes FileReader and BufferedReader, which allow the text from a file to be read line by line. When my system receives a file from the user, it starts with reading the first line, and checks whether the first characters correspond to the word “digraph”. If this is not the case, it stops immediately, because this means that this file does not describe an automaton (see next section). Rejecting a file if its beginning is not correct saves some time, we do not read all the file before starting to look at what it contains. If it is not rejected at this point, then we transform the whole text in a String, and give it to the parser.

To create a file and write in it, I used the class PrintStream from Java libraries. It provides methods to open a stream and to input text in the file. This way, automata created by the program can be saved, with a view to reusing them later.

4.3.2 Parsing DOT files

After loading a DOT file, the program has to create an automaton from the description it contains. To this end, a parser was implemented. This parser takes as argument a string corresponding to the content of the input file, and tries to convert it into an automaton. But this is not always possible: indeed, if the file is not a correct description of an automaton following DOT format specifications, the construction will not succeed. There are three cases we could encounter:

- The string is not a correct DOT file, as specified in GraphViz documentation [9];
- The string correctly describes a graph in DOT format, but this graph does not correspond to a finite automaton in the way I chose to describe it;
- The string corresponds to the description of a finite automaton.

In the first and second cases, no automaton will be created. Only the third case is a valid one. An example already given in Section 3.3 shows a good description of an automaton.
A token is defined here as being either an identifier, that is a string containing only letters (uppercase or lowercase) and digits, or another single character (except from “->” which is a 2-characters token). Spaces, tabulations or newline characters are not considered as tokens, but it is possible to include some at the beginning or at the end of the file, as well as between any two tokens (for presentation purpose).

First the parser has to make sure that the first token is the word “digraph” and that the second one is a name for this graph (an identifier). The rest of the file should be surrounded by brackets “{” and “}”, and each instruction between those brackets is analysed. An instruction is the text contained before the first semicolon, or between two semicolons. The parser classifies instructions into six categories, depending on their content:

1. **the first instruction for initial state**, that has to be exactly “start [style=invis, shape=point];” except from the facts that tokens can be more or less spaced, and the options `style` and `shape` can be exchanged;

2. **the second instruction for initial state** of the form “start->q0;”, where q0 can be replaced by any other identifier;

3. **instructions of final state** of the form “q4 [peripheries=2];” where q4 is replaced by any identifier;

4. **instructions of state definition** of the form “q5;” with any identifier before the semicolon;

5. **instructions of transition** of the form “q0->q1 [label="1"];” where q0 and q1 are any identifiers, and the `label` is equal to an identifier of length 1;

6. **bad instructions**: those are the instructions that do not go in any of the previous categories. If there is such an instruction, the file is immediately rejected.

After reading the whole file, we should have seen both instructions for the initial state, and at least one instruction of final state, otherwise the file is rejected. The instructions do not have to follow a fixed order. At that point, if the file is correct, an automaton is constructed as follows:

- A set of states is created. For each state name specified in an instruction (this can be done in the second instruction for initial state, in instructions of final state, instructions
of state definition or instructions of transition), a new state is added to the set. If the same name appears several times, it will correspond to a unique and same state;

- The name in the second instruction for initial state corresponds to the initial state;
- The names in instructions of final states correspond to the final states;
- Transitions are created following the instructions of transition. The label is the symbol of the transition.

Apart from this parser, a method was implemented to transform an automaton into its description in DOT format. This is somehow the other way around. Thus an automaton obtained after applying methods from the software can be saved, and possibly reused later.

### 4.3.3 Graphical user interface

The final step to bring together all the previous implementations is the construction of a graphical user interface. The goal of this stage is to make the software system easily accessible, by providing an intuitive interface. However, this system is intended for users who are already familiar with finite automata: a nice GUI will not change this fact. Here the graphical functioning of the software is described, and then I explain how I coded the GUI using Swing.

The final version of the software system has a graphical window, with two main areas to display automata (see screenshot in Section 3.4). On each side, we can load a finite automaton from a DOT file using the buttons at the bottom, for browsing the file system. At any time, those two automata can be exchanged using the button at the top. Under each of them, there is a little text area displaying information, in particular their size. The automaton on the left is the main automaton, the one we can operate on. When a method is applied to this automaton, the result will be displayed on the right. But we can also open any other automaton on the right, to test its equivalence with the main one. The possible actions to apply on those automata can be chosen from the menu at the top of the window. This menu contains most of the methods implemented and previously described in this chapter, but not all of them, in particular:

- Some simple actions such as testing if an automaton is a DFA, or testing if it accepts a string, have not been included. This is because those are not related to the goal we pursue, that is reducing NFAs;
- The enumeration of NFAs is absent as well, the reason being that this method is not achievable in most cases, as explained earlier.
This graphical interface is created using Swing (introduced in Section 3.4). The Swing components mainly used are the following:

- **JFrame**: the main window of the program;
- **JPanel**: used as the basic container to include other elements;
- **JLabel**: to display pictures (the automata) or text (their description);
- **JMenuBar, JMenu and MenuItem**: elements to build the menu whose items are the methods that can be applied;
- **JButton**: to represent the buttons;
- **JFileChooser**: a graphical tool to browse the file system, and thus opening or saving files;
- **JOptionPane**: a dialog box to display information, for example when a wrong file has been indicated;

All those elements are included in a class `GraphicWindow` I created. This class implements the interface `ActionListener` provided by Swing, in order to “listen” to the objects and define for each button and each menu item a specific action: the corresponding methods are invoked to produce the expected outputs. At all times, an instance of the class `GraphicWindow` possesses a current automaton `current` and another one `other`, as well as two strings corresponding to their description in DOT format, in order to be able to display their pictures. Those automata can be equal to `null`, and their description be the empty string, if they have not been defined or computed yet. Their pictures are displayed thanks to GraphViz Java API [1] which calls the program `dot` to create a temporary file containing the image, and this file can then be opened in the graphical window. When a user opens a DOT file, `current` or `other` (depending on the chosen side) becomes the automaton corresponding to this file, and its picture is computed and displayed. When he chooses from the menu or the buttons an action to operate on `current`, the output of the invoked method is stored in `other` and the associated DOT description is updated, so that the picture can be renewed as well.
Chapter 5

Evaluation

The evaluation appears to be a crucial point of this project, in the sense that several methods working on NFAs have been enlightened, but their real efficiency is not proved. Indeed, for the moment the only trials consisted in finding some nice examples, on which those methods gave good results, but those are maybe not representative of all kinds of automata. In order to evaluate the practical efficiency, the main idea here is to generate some random automata, apply the different methods on them and compare the size of the resulting NFAs. But first, an overall evaluation of all the aspects of the project.

5.1 General evaluation of the project

In this first approach, the criteria to evaluate the project are the relevance of the background reading, the correctness of the algorithms implemented, the ease of use of the software and the suitability of the initial schedule. In the next section, the efficiency will be tested.

The background reading relied mainly on two types of sources: textbooks on automata theory and languages, and research articles. This combination allowed me to deal with the usual background on finite automata and to compare how this was tackled in the different books, but also to be aware of various approaches of NFA reduction. The reading work started by recalling all the definitions thanks to the textbooks, and it continued with exploring the possibilities of reducing NFAs given by the papers selected. This was not a linear pattern though, because while dealing with the articles, I often needed to go back to textbooks. In addition, references on Java languages helped me for the implementation work. So I think that the background reading was relevant. The only thing is that it could have been wider, especially
concerning the use of automata in real applications.

I made sure of the correctness of the algorithms implemented thanks to two stages. First, I followed exactly the descriptions of the algorithms in the research papers they appeared. For algorithm not described in any article (for example the enumeration of NFAs or the reverse method), I started writing the algorithms formally before implementing them. This stage made it possible to have a good overview of a method prior to implementation, and thus the coding became very easy, simply translating English sentences into Java language. Then, to test the methods, I created examples of automata (in DOT format) to compare the executions of my methods with those of JFLAP. But this was possible only for methods that I had in common with JFLAP, that is, the usual operations on finite automata: determinisation, minimisation, etc. The different examples were constructed with different characteristics, in order to test the methods on the maximum of possibilities. All tests I have run this way worked well, the results were always the same as in JFLAP. In conclusion, we can be certain of the correctness of the methods implementing the algorithms.

The graphical user interface meets the objective of providing an easy way to interact with the system. Indeed, it has clear menus where the operations are separated between those that are intended to reduce NFAs and the others. Moreover, there is a third menu were all other actions are classified: opening or closing a file, and quitting the program. The two areas for the automata are well defined, and each of them has its own button to input an automaton. Basic information is provided at their bottom. When we apply methods on them this information is updated, and sometimes a dialog box appears to inform of a bad operation or of the result of an action. This software system is thus rather simple and intuitive to use.

The schedule chosen at the beginning of the project (see Section 1.6) appeared to be a good one, as I followed it almost exactly. The only thing is that the background reading, although being the most intensive on the indicated period (week 4 to Christmas break), continued a little after that period, until the end of the project. It was necessary to keep an eye on resources while programming, for this project both had to be considered together. Finally, this was a good idea to allow plenty of time to write the final report (five weeks).
5.2 Randomly generated automata

5.2.1 Idea

The idea is based on the comparison of the size of randomly generated automata with their minimum DFA, the NFA found by the state equivalence method and the NFA found by the reversal method. Working with random automata is a good way to get objective statistics on the efficiency of those methods.

5.2.2 Creation of random automata

An extension to the program has been made in order to test the different methods to reduce NFAs. It is an extra file, still in Java language, that uses the functions previously implemented. This part of the program generates random automata, so as to test the reducing methods on them. The idea of a random model to generate NFAs has been explored in [18] and [5]. Drawing inspiration from those papers, and also reusing part of the implementation done for the enumeration of NFAs (Section 4.2.7), I was able to build my own random model.

A function generates an NFA for given number of states, transitions, final states and symbols in the alphabet. First the set of states and alphabet of the automaton are created, and then the final states have to be selected. To this end, the Java class Random, which generates pseudorandom numbers, is used: it chooses numbers, representing state IDs, and the states corresponding to those IDs will be final. The same trick is used for the choice of transitions, but we had first to define and order on all possible transitions of the automaton. This was done by creating an array of size \( n^2k \) where \( n \) is the number of states and \( k \) the size of the alphabet, and each element is a possible transition. Then a Random element chooses the transitions to include in our automaton.

5.2.3 Results

Experience 1

The first approach for evaluating the different methods with help of the random model was, from many random finite automata with fixed number of states, transitions, final states and with a given alphabet, to look at the average size of automata after applying each method. I decided to fix the alphabet with only two symbols (‘0’ and ‘1’) because this is not the most important criterion to consider, and a binary alphabet is sufficient to demonstrate the capacities of finite
automata. Then I made the number of states vary, to observe the differences we obtained. As it appeared that the ratios \( \frac{\text{number of transitions}}{\text{number of states}} \) and \( \frac{\text{number of final states}}{\text{number of states}} \) did not play a crucial role in the results, I present here what we obtain for approximately 2.5 transitions per state and 40% of final states. Those seem to be good proportions, as a random automaton will then have a good probability of being connected, and the language accepted is likely to be neither the empty set nor the set of all strings on the alphabet. I have been running some tests on generated automata with 5 to 35 states. For a fixed number of states, 100 automata are generated, and three methods operate on them: minimum DFA, reverse method and state equivalence method. For each of those methods, the average number of states of automata thus obtained is calculated. Graph 5.1 shows the evolution of this average number of states against

Figure 5.1: Number of states of finite automata after applying reducing methods
the size of input automata. The blue curve called “input automaton” represents the number of states of the input, it is only a linear function to compare the others with.

**Result and interpretation**

We notice that the minimum DFA is much bigger on average than the input NFA, its curve seems to grow exponentially. This confirms what was said about DFAs being exponentially larger than NFAs. The state equivalence method actually works well: the size of automata obtained this way are on average 23% smaller than their input, and this percentage seems to stay constant as inputs get bigger. On the contrary, the reverse method does not seem to have good results, and gives on average much bigger automata than the input: its curve is exponential and follows almost exactly the curve of minimum DFA, even exceeding it a bit. But this is in fact understandable: this method does not aim to produce a smaller NFA each time, but it constructs an automaton with the same number of states as the minimum DFA of the reverse (or one more), that can possibly be smaller than the input. That is why the average size is equivalent to the size of minimum DFAs.

**Experience 2**

Instead of focusing on the average number of states, another approach is now to examine the cases where the reverse method gives the best result. A function was implemented that, from randomly generated automata as described above, return the automata on which the reverse method works better than the two other methods, that is the state equivalence and the minimum DFA. Like in the previous experience, NFAs are randomly generated for a given size, and the size varies between 5 and 25 states (25 was the maximum for this test to be ran in a reasonable time). But this time, we generate 1000 automata for each size. We observe the rate of NFAs for which the reverse method is the best reducing method.

**Result and interpretation**

It appears that the proportion of automata reduced by the reverse method is lower as the number of states increases. The results obtained are summarized in this table:

<table>
<thead>
<tr>
<th>Number of states of the input</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cases where reverse method is best</td>
<td>20.9%</td>
<td>10.3%</td>
<td>7.1%</td>
<td>4.6%</td>
<td>2.2%</td>
</tr>
</tbody>
</table>
Moreover, in most of those cases, the minimum DFA already gives a really good result, and the NFA obtained by the reverse method only have one or two states less than the minimum DFA. We observe that the reverse method gets good results when there exists a much smaller automaton equivalent to the input, and that is probably why it works best on NFAs that are already small.

5.2.4 Conclusion and limits of randomly generated automata

Thanks to this study, we have seen that the state equivalence method is probably the best one we have found in this project: it operates constant reductions on NFAs, whatever their size is. The reverse method is also interesting but it should be used on small NFAs only, as its efficiency decreases for bigger automata. Finally, the minimum DFA stays the smallest automaton that can be found in a few cases. In conclusion, the best way to reduce NFAs is to apply all those methods and take the smallest NFA found, because depending on the case, this is not always the same method that will give the smallest one. And this evaluation did not allow me to determine the criteria of an automaton that would make one method work best on it.

A potential issue that can be pointed out is that random automata may not be representative of the automata actually used in applications. Maybe the automata used have some common characteristics that make techniques such as the reverse method or the equivalence of states work better or worse on them. From this point of view, this type of evaluation reaches its limits. Tabakov and Vardi [18] have raised that issue in their article, and they say that it is not really possible to know to what extent their probabilistic model is realistic because of the lack of benchmarks of finite automata. But they also say that it is difficult to see how this model could be contradicted, so there is no reason not to take the results obtained into account. To explore further in that direction, this would be useful to focus on a particular application of finite automata, and to have an idea of the kind of automata that are used for it.
Chapter 6

Conclusion

6.1 Meeting objectives

The minimum requirements have been successfully accomplished. Here is a description of how this was done:

- **Convert an NFA into an equivalent DFA**: the subset construction has been studied and efficiently implemented to turn NFAs into DFAs;

- **Given a DFA, find the minimum equivalent DFA**: several minimisation algorithms have been considered, and Hopcroft’s algorithm was implemented and included in the software;

- **Check whether two NFAs are equivalent**: this was done by computing the minimum DFAs corresponding to those NFAs, and checking if they were isomorphic by finding an appropriate one-to-one mapping between their states;

- **Given a DFA accepting \( L \), construct an NFA accepting the reversal of \( L \)**: the reverse automaton is computed by reversing all transitions, and exchanging initial and final states;

- **Given a DFA accepting \( L \), construct an NFA accepting the complement of \( L \)**: the construction of the complement of a DFA was simply implemented, consisting only in exchanging final and non-final states;

But those requirements were even exceeded. Methods to actually reduce NFAs were found and implemented (enumeration of NFAs, reverse method, state equivalence method), and a
graphical user interface was constructed. So the main goal of this project was reached, that is creating a software system able to reduce NFAs. However, this cannot be achieved for all inputs, and evaluation showed that only the state equivalence method is in fact really efficient. But this is understandable as the minimisation of NFAs is still an open problem.

6.2 Further work

Here are other methods to compute smaller NFAs that could be examined and, where possible, implemented. Those ideas are either discussed in research papers or my own:

- **Extending state equivalence**: The state equivalence $\equiv_R$ described in Section 2.3.3 was from an article by Ilie and Yu [13]. They also define another equivalence $\equiv_L$ in this paper, detailed in a later one [14]. This new equivalence is symmetric to $\equiv_R$, which only considered distinguishability of states to the right. Thus equivalent states with respect to $\equiv_L$ can also be merged without changing the language accepted. The implementation of $\equiv_L$ would only require to apply $\equiv_R$ to the reverse automaton. The combination of those two equivalences could result in a more efficient reduction, but the way this should be done is not known;

- **State merging by other means**: Champarnaud and Coulon [4] introduce the possibility of reducing NFAs by a state merging method based on preorders instead of equivalences. They say that this method is more efficient than the state equivalence we implemented;

- **Finite languages**: It could be interesting to consider specifically automata accepting languages that contain a finite number of strings. Minimum finite automata for those languages would be trees, i.e. they would not have loops, otherwise it would possible to go around these loops as many times as we want. Indeed, in an automaton representing a finite language, all the states contained in a loop (and their successors) could be removed, because they would not be co-accessible. We can also eliminate the leaves that are not final states, for this same reason. Then, it would be very easy to determine the *right* and *left languages* (defined in [4] for example) of each state, with a view to merge the states having one of these languages in common.

Another direction that was not really explored in this project is testing the efficiency of applying several methods on an NFA, one after another. It could be possible to determine to
what extent this would reduce more the NFA, and we could look for the best order in which to apply those different methods.

Apart from the experimental evaluation of methods to reduce NFAs (Section 5.2), we could investigate their space and time complexity. Most of the papers mentioned in this report take this approach.

It would also be interesting to use reduction of NFAs to compute smaller regular expressions, as proposed in [10]. There are known constructions to build a finite automaton from a regular expression, and the other way around, and their size are closely related.

Concerning the necessary conditions for using the software produced, detailed in Appendix D, I could try to eliminate some of them to extend the possibilities of use. This could be done by making the system available for more platforms (Windows, Mac OS, ...) and by finding a way to avoid the prior installation of dot as a requirement.

Finally, as suggested in Section 3.3, the software could take inputs or produce outputs in XML format. This would make it compatible with JFLAP.
Bibliography


Appendix A

Personal reflection

During the year, I had to put a considerable amount of work and effort in the completion of this project. Now, I think that I am satisfied with what have been achieved. This was the most challenging work I have ever done during my time at university. Although there have been some difficult times, this was overall a good experience. But I must say that I am relieved this is over!

At the beginning of the project, I had great expectations: I thought that I would be able to find a lot of techniques to reduce NFAs in articles, or even imagine some myself, and choose the best ones to be implemented. The first obstacle to my enthusiasm was the limited number of research papers on that particular subject, as well as their complexity. So I think that while dealing with open problems in their projects, students should be aware that it might be difficult to find appropriate literature, and thus determine precisely objectives that can be achieved in the duration of the project.

I should have spent more time on reading. Since my project consisted for a half in research, and the other half was implementation, the reading stage was definitely the most important stage. I do advise students with a similar type of projects to consider this. I think that one of my errors was to rush through initial reading, and when I started the implementation I realised that I was lacking some knowledge. Taking time to read and to really understand all the papers that I have found would have given me the opportunity to explore further the existing methods to reduce NFAs. Maybe then I would not have had the time to implement them all, but I would have been able to choose the best ones among those methods: this could have kept me from coding a method like the enumeration of NFAs, which ended up not being very useful. But the
reading should not be done exclusively prior to coding, it is also necessary to keep on reading papers during the implementation: those two aspects are complementary.

It is very important to take notes all through the duration of the project, to be sure not to forget things. In particular, after each meeting with my supervisor, it helped me a lot to write what had been discussed and to keep these notes. Concerning information discovered while reading, one should always include in their notes the corresponding references (together with the page number, or section). Many times, I had to search again in several textbooks for something that I remembered having read, but I could not remember where.

Writing this report was a long and difficult task, especially because English is not my mother tongue. As an international student, this gave me a great opportunity to develop my writing skills in English. I hope I did not make too many mistakes... I advise exchange students to make sure they are strongly motivated before choosing to undertake a project, because this really is a lot of work. It was definitely a good idea to allow plenty of time for writing the final report (5 weeks), this gave me the time to take care of phrasing everything properly. Producing the report also improved my \LaTeX knowledge, including PSTricks to draw graphs.
Appendix B

External material

The external material used for this project was:

- Java standard libraries [15], in particular elements of the following packages: java.util, java.io, javax.swing and java.awt;

- GraphViz [9], to run dot, allowing the picture of the graphs to be displayed;

- GraphViz Java API [1], providing (among other things) a method to call dot from a Java program and return the picture created;
Appendix C

Ethical issues

There were no ethical issues associated with my project, as I did not use personal data or anything else that could harm others.
Appendix D

Requirements and advice to use the software

The requirements for a machine to be able to use fully the software system are:

- A UNIX platform, because of the way dot is called and the location of temporary pictures. This could be extended to Windows for example, by specifying the right locations for this program;

- Java has to be installed on the machine, this is essential to be able to execute the program;

- Dot program should also be installed, because it is necessary to display the pictures. But we could imagine a different way to create those pictures, where installing dot independently would not be necessary anymore.

The CD provided with this report contains the software system, as a JAR archive, and a directory containing examples of DOT files. The software can be executed in command line by `java -jar ReduceNFAs.jar`, or simply by a double-click on it.