Bachelor thesis
Computing Science

Radboud University

The Practical Performance of Automata Minimization Algorithms

Author:
Erin van der Veen
s4431200

First supervisor/assessor:
dr. J.C. Rot
`j.rot@cs.ru.nl`

Second assessor:
prof. dr. J.H. Geuvers
`h.geuvers@cs.ru.nl`

August 7, 2017

Abstract

Given an automaton A, automata minimization is the problem to create an automaton with the same language L, such that there is no automaton with less states and language L. There has already been some research into the practical performance of automata minimization algorithms. Many of these experiments, however, use some kind of random automata generator to resemble practical automata. In this thesis Hopcroft's and Brzozowski's algorithms are benchmarked on a benchmark set acquired using intermediate steps of model checking. This provides a better understanding of the performance these algorithms could have in practice. The results show that Hopcroft's algorithm clearly outperforms Brzozowki's algorithm in nearly all cases. In order to provide a fair comparison, the algorithms were also benchmarked on random automata. The results of this benchmark are in line with the results of the experiments mentioned earlier.

Contents

Chapter 1

Introduction

Automata theory is a branch of computing science that studies the theory and applications of automata. Automata are basic abstract models of computation that recognize some language. A basic problem in automata theory is finding the minimized form of an automaton. A minimized automaton is an automaton with the same language and a minimal number of states.

Automata minimization is used in many practical applications, and as such, it is desired that the complexity of running these automata is kept at a minimum. There are several algorithms that are used for minimization. The best known (and understood) algorithms are arguably Hopcroft's [7], Brzozowski's [5] and Moore's [9] algorithms. The theoretical performance of these algorithms is well understood, but not much is known about their practical performance [2]. It should be noted that the practical performance of automata minimizing algorithms need not be the same as the average theoretical performance, as the automata that are used in practice might not be equally distributed among all possible automata. Almeida et al. [2], Watson [12], and Tabakov and Vardi [11] tried to find out which algorithm performs best in practice. All these attempts use randomly generated automata. Randomly generated automata might not accurately represent automata that are used in real world applications, since there might be some characteristics of automata that are more frequent than others in the real world. These characteristics would not be accounted for by a random generator.

The results of Watson show that Brzozowski's algorithm performs better than Hopcroft's algorithm, which is unexpected when considering the exponential versus logarithmic worst case complexities. This thesis sets out to support or disprove that result by using a benchmark set instead of randomly generated automata.

The benchmark set used in this thesis was obtained from the "intermediate steps of model checking" [1] with the purpose of being used in a benchmark. While these automata might not accurately represent all automata that are used in practice, they could still provide insight into possible differences between the performance of the algorithms when used on random automata, versus practical automata. In order to provide a fair comparison, the benchmark set will be compared to randomly generated automata. The random automata are generated using a method described by Tabakov and Vardi [11] and both the benchmark set and the randomly generated automata will be tested on the same implementations.

As described in Chapter 6, the results of my research show that Hopcroft's algorithm outperforms Brzozowski's algorithm in nearly all tested automaton. These results are in line with the results obtained by Almeida et al. [2], and Tabakov and Vardi [11].

This thesis is structured as follows. The second chapter contains the preliminaries and introduces all notation used in the rest of the thesis. The third and fourth chapter introduce Brzozowski's and Hopcroft's algorithms respectively. They also provide a basic overview and highlights of the implementation. Chapter 5 describes how the benchmarks in this thesis were performed. Chapter 6 focuses on the experimental results that were obtained in similar research and provides a comparison between their results and the ones acquired in this research.

Chapter 2

Preliminaries

2.1 Deterministic Finite Automata

In order to present the automata minimizing algorithms in this thesis, it is useful to introduce a consistent notation and basic principles of automata minimization. This section serves to provide both.

A Deterministic Finite Automaton (DFA) is a 5-tuple $(\Sigma, S, s_0, \delta, F)$ where:

Σ is an *alphabet*; a finite set of *letters*

S is the finite set of states

s_0 is the initial state such that $s_0 \in S$

δ $: S \times \Sigma \rightarrow S$ is the transition function

F is the set of accepting states such that $F \subseteq S$

A *word* is a finite *string* of letters, where a string is an ordered list of letters. The empty word is denoted as ϵ. Given an alphabet Σ, the set of all possible words is Σ^*. The *δ-function* (transition function) is a binary function that maps some letter and a state to some state. A state s *transitions* to s' with letter l if $\delta(s, l) = s'$.

The δ-function is overloaded on words such that the signature changes to $\delta : S \times \Sigma^* \rightarrow S$ where it represents the state that the δ function results when it is applied to every letter in a word consecutively. This is recursively defined as:

$$\delta(s, \epsilon) = s$$
$$\delta(s, lw) = \delta(\delta(s, l), w)$$

The *inversed δ-function* ($\delta^{-1} : S \times \Sigma \rightarrow \mathcal{P}(S)$) is also defined. Given a state s and a letter l, δ^{-1} serves to return a set of states S' that contains all states that transition to s with l: $\delta^{-1}(s, l) = \{s' \mid \delta(s', l) = s\}$.

A state s in an automaton $A := (\Sigma, S, s_0, \delta, F)$ is said to *accept* a word w if $\delta(s, w) \in F$. Every state s in automaton A has a language $\mathcal{L}(A, s)$ defined as:

$$\mathcal{L}(A, s) = \{w \in A^* \mid \delta(A, s) \in F\}$$

The language $\mathcal{L}$ of an automaton A is the set of words that is accepted by the initial state of the automaton: $\mathcal{L}(A, s_0)$. We denote $\mathcal{L}(A, s_0)$ by $\mathcal{L}(A)$.

A *minimal* DFA A with states S is such that there is no other DFA A' containing states S' where $\mathcal{L}(A) = \mathcal{L}(A')$ and $|S'| \leq |S|$. Note that a minimal automaton A of some language $\mathcal{L}(A)$ is unique in the way that there are no other automaton that accepts the same language with the same amount of states [8].

2.2 Non-deterministic Finite Automata

A Non-deterministic Finite Automaton (NFA) is an extension of notion of DFAs that, in essence, extends the δ-function. Where a DFA has the δ-function defined as $\delta : S \times \Sigma \to S$, an NFA has it defined as $\delta : S \times \Sigma \to \mathcal{P}(S)$, this enables a single state to transition to multiple states with a single letter. Furthermore, a common extension is to also allow multiple initial states. As such an NFA is defined as $(\Sigma, S, I, \delta, F)$ where δ is the aforementioned function, and I is the set of initial states where $I \subseteq S$. The rest of the five-tuple remains as it was.

We need to update our definition of languages to match with the new type of the δ-function. For an NFA A, the language is defined as: $\mathcal{L}(A) = \{w \mid \exists_{s \in I}[\delta(s, w) \in F]\}$.

2.3 Determinization

Let A be an NFA, the act of determinization is constructing a DFA A' such that $\mathcal{L}(A) = \mathcal{L}(A')$. A typical method of doing this is through "Subset construction". The intuitive idea is that we represent a set of NFA States as a single DFA state. The pseudocode clarifies:

Consider the NFA in Figure 4.2 on page 25. Applying subset construction yields the automaton in Figure 4.3. For clarity, the DFA states are labeled with the NFA states that they represent.

2.4 Reversal

Let $l_0 l_1 l_2 \ldots l_{i-1} l_i$ be some word w where l_i is the i'th letter of a word. The *reversed* word w^R is then defined as the word where all letters are in reverse order: $l_i l_{i-1} \ldots l_2 l_1 l_0$.

Algorithm 1 Subset Construction

```
Require: $NFA := (\Sigma, S, I, \delta, F)$
Ensure: $DFA := (\Sigma, S', s'_0, \delta', F')$ such that $\mathcal{L}(NFA) = \mathcal{L}(DFA)$
1: Create the initial states of the DFA that represents $I$
2: for the newly created DFA State do
3:     for all $a \in \Sigma$ do
4:         Apply $a$ on the state to obtain a new set of states
5:         This set is a (new) DFA state
6:     end for
7:     Apply step 2 to every new DFA state
8: end for
```

The reversed language $\mathcal{L}^R$ of some language $\mathcal{L}$ is the language where all words are reversed:

$$\forall_{w \in \Sigma^*}[w \in \mathcal{L} \Leftrightarrow w^R \in \mathcal{L}^R]$$

Given a DFA (or NFA) A, a NFA A' can be constructed that accepts the reverse language. When A' is constructed, we say that A is *reversed*. A DFA can be reversed using the following algorithm:

Algorithm 2 DFA Reversal

```
Require: $DFA := (\Sigma, S, s_0, \delta, F)$
Ensure: $NFA := (\Sigma, S, I, \delta', F')$ where $\mathcal{L}(NFA) = \mathcal{L}(DFA)^R$
1: Reverse the delta function, iff: $\delta(s, a) = s'$, then $s \in \delta'(s', a)$ for every
   $S \times l$. This can also be written as: $\delta' := \delta^{-1}$.
2: $I = F$
3: $F' = \{s_0\}$
```

2.5 Equivalence Classes

To understand the notion of an equivalence class, one must first understand what an equivalence relation is, which in turn requires understanding of relations. A *relation* R is a set of tuples over a set X ($R \subseteq X \times X$). An *equivalence relation* is a relation that is binary, reflexive, symmetric and transitive.

The properties of an equivalence relation ensure that within a domain, subsets must arise for which all elements are in the equivalence relation with each other. Such groups are equivalence classes.

Definition 1. *Let R be an equivalence relation, and x be some element. The equivalence class of x with relation R is then defined as:*

$$[x] = \{y \mid xRy\}.$$

Definition 2. *A partition P of some set X, is defined by the following properties:*

- $P \subset \mathcal{P}(X)$
- $\forall_{C,C' \in P}[C \neq C' \rightarrow C \cap C' = \emptyset]$
- $\bigcup_{C \in P} = S$

The number of classes that a partition P contains is denoted as $|P|$.

Since no element of a set X can be in two different equivalence classes with the same relation R, the equivalence classes of R define a partition P of X. This is denoted as $P = X_R$. Conversely, any Partition P on X defines an equivalence relation R.

These concepts can be applied to automata theory. Consider some automaton $(\Sigma, S, s_0, \delta, F)$. In order to create the aforementioned equivalence classes on states, the equivalence relation must first be defined. We use "language equivalence" for this purpose. If two states have the same language, they are "language equivalent", and therefore in the same equivalence class.

Definition 3. *Two states s, s' in an automata A are language equivalent iff:*

$$\mathcal{L}(A, s) = \mathcal{L}(A, s')$$

This thesis uses the notation $s \sim s'$ to denote that that s and s' are language equivalent.

Consider the automaton in Figure 2.1. The first state has the language that is represented by the regular expression $(a|b)a^*b(a|b)^*$, it is the only state that has this language, and is therefore in an equivalence class of its own. Another equivalence class contains state 2 and 3, both having the language represented by $a^*b(a|b)^*$. State 4 is in the last equivalence class. The complete partition is: $\{\{1\}, \{2, 3\}, \{4\}\}$. For readability the rest of this thesis will use the following notation: $1 \mid 2, 3 \mid 4$.

2.6 Partitions to DFA

Definition 4. *Let $A := (\Sigma, S, s_0, \delta, F)$ be some DFA and P be the partition $S_\sim$. We can then construct the new (minimal) automaton A' as $(\Sigma, S', s_0', \delta', F')$, where:*

S' is P

s_0' is the class $C \in P$ for which $s_0 \in C$

δ' : $S' \times \Sigma \rightarrow S'$ such that $\forall_{s \in S, l \in \Sigma}[\delta'([s], l) = [\delta(s, l)]]$

F' be a set such that

$$\forall_{C \in P}[\exists_{s \in C}[s \in F] \Leftrightarrow C \in F']$$

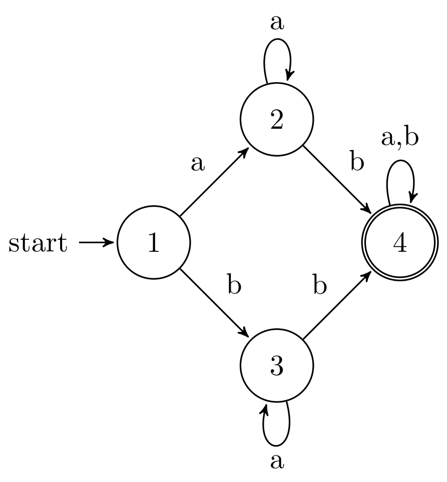
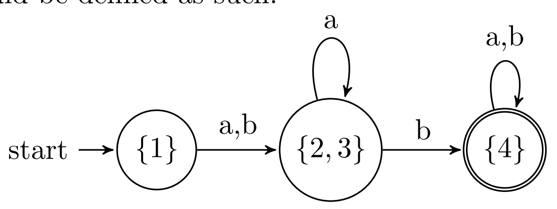


Figure 2.1: Small Example Automaton

When a partition based on language equivalence is created for some automaton A, it can be used to construct a minimal DFA A' where $\mathcal{L}(A) = \mathcal{L}(A')$. The idea rests on the fact that a state s with transition $\delta(s, a) = s'$ could instead have a transition to any other state in the equivalence class of s'. As such, we can create a new DFA where we consider every class in the partition to be a single state.

Proof. Let A be some automaton with s and s' being states in that automaton. Let C be the equivalence class based on that contains at least s' and s''. Given the transition function $\delta(s, a) = s'$ it is trivial that $\{a\} \cdot \mathcal{L}(A, s') \subseteq \mathcal{L}(A, s)$ where $\cdot$ is the concatenation function. If we now assume that $\delta(s, a) = s''$ instead, we can trivially say that $\{a\} \cdot \updownarrow(s, s') = \{a\} \cdot \updownarrow(s, s'')$. Given this, it is shown that $\{a\} \cdot \mathcal{L}(A, s'') \subseteq \mathcal{L}(A, s)$. As such, we can conclude that any transition $\delta(s, a) = s'$ can be replaced by $\delta(s, a) = s''$ if s' and s'' are in the same equivalence class. Which, in turn, shows that equivalence classes can be considered as single states without changing the language of the automaton. $\square$

Consider the automaton in Figure 2.1 with partition $1 \mid 2, 3 \mid 4$ as an example. Following the definitions above a new automaton $(\Sigma, S, s_0, \delta, F)$ would be defined as such:



When a DFA is created in such a way, it is guaranteed to be the minimal automaton.

2.7 Partition Refinement

Given a partition P, a refined partition P' is such that the following holds:

- $\bigcup_{C \in P} = \bigcup_{C' \in P'}$
- $|P| < |P'|$
- $\forall_{C' \in P'}[\exists_{C \in P}[C' \subseteq C]]$

In this case, P' is a *refinement* of P, vice versa, P is *coarser* than P'.

Chapter 3

Hopcroft's Algorithm

This section first introduces the concept of using partition refinement in DFAs, followed by an overview of Moore's algorithm as it provides a solid basis for Hopcroft's algorithm. An overview is then given on Hopcroft's algorithm. Lastly, the differences between Moore's algorithm and Hopcroft's algorithm are highlighted, and a reasoning is provided that Hopcroft's algorithm is correct, provided that Moore's algorithm is correct. This is done by individually considering the correctness of all aforementioned differences.

3.1 Partition Refinement in DFA Minimization

Moore's and Hopcroft's algorithm share a very similar approach to automata minimization [12], utilizing equivalence relations, partition refinement and having a top-down approach. Given an automaton $(\Sigma, S, s_0, \delta, F)$, the goal of the algorithms is to create a partition P of S of which the associated equivalence relation is language equivalence $(\sim)$. Once this partition is created, the minimal DFA can be constructed using the algorithm described in the preliminaries. In order to create the partition $S_\sim$, partition refinement is used to gradually refine the coarsest possible non-trivial partition $\{S-F, F\}$ until it equals $S_\sim$.

Splitters A *splitter* is a concept used in both Hopcroft's and Moore's algorithm. It is a tuple (S', l) of a set of states S' and a letter l, that is used in refining a partition. In particular, it is used to split a set S of states into two new sets S'' and S'''. Moore's and Hopcroft's algorithm do this by checking for every element in S, if it does or does not have a transition to S' with l. All states that do are then separated from those that do not, such that either $S'' = S' \cap \bigcup_{s' \in S'} \delta^{-1}(s', l)$ and $S''' = S' \; \bigcup_{s' \in S'} \delta^{-1}(s', l)$, or $S'' = S' \; \bigcup_{s' \in S'} \delta^{-1}(s', l)$ and $S'' = S' \cap \bigcup_{s' \in S'} \delta^{-1}(s', l)$ holds.

A set S is *split* if both S'' and S''' are non-empty. We denote this as: $(S'', S''') = S \mid (S', l)$. A special case is where a set S forms a class in a

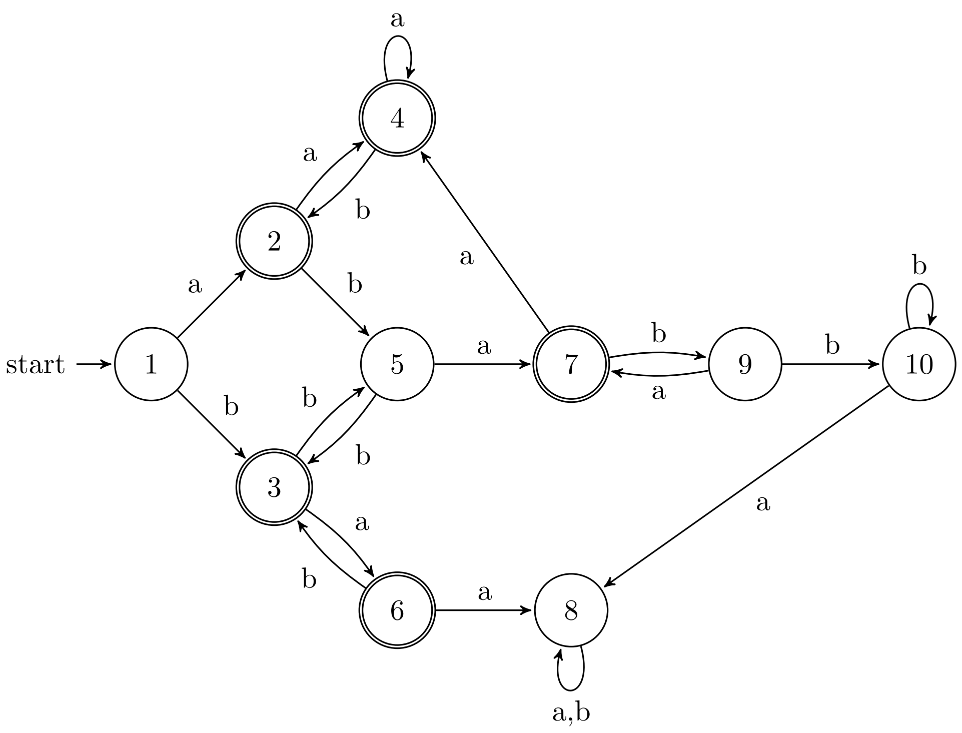


Figure 3.1: Automaton 1 [6]

partition P, we then say that the set S is *split* in P if the set is replaced in the partition with the two new sets. Note that replacing a class in a partition as described does not violate any of the properties of a partition.

Moore's Algorithm

Moore's algorithm was initially defined for Moore Machines [9], but since DFAs are a special case of Moore Machines, the algorithm is also applicable on them. I will first introduce a general overview on the algorithm, and will then provide an elaborate example. The pseudocode of the algorithm can be found in Algorithm 3.

Algorithm 3 Moore's Algorithm

```
Require: $A := (\Sigma, S, s_0, \delta, F)$
Ensure: The minimized automaton with the same language as A
 1: $P = \{F, S - F\}$
 2: repeat
 3:    $P' = P$
 4:    for all $C \in P'$ do
 5:       for all $l \in \Sigma$ do
 6:          for all $C'$ split by $(C, l)$ do
 7:             Split $C'$ in $P$
 8:          end for
 9:       end for
10:    end for
11: until $P == P'$
12: return a new DFA based on $A$ and $P$
```

Let us start with a global overview of the algorithm. The algorithm starts with the aforementioned coarse partition $P = \{F, S - F\}$. Every iteration of outermost for-loop Moore's algorithm stores the current partition in a variable P'. This allows the partition that needs to be refined to be destructively updated while still allowing access to the partition as it was before the iteration.

Every class of P' is then used in combination with every letter to create a splitter s. This splitter is then used to refine partition P if possible. There are many implementations possible, but the most typical way is through the inverse δ-function outlined in the preliminaries. First, it creates a set I where $I = \bigcup_{s \in C} \delta^{-1}(s, l)$. Then, it splits every class with an element (but not all) in I. If, after a complete iteration over the classes of P', P has not been updated, the algorithm converts the partition P to a DFA. This DFA is then returned.

Using this overview, let us consider the behaviour of Moore's algorithm when run on the automata specified in Figure 3.1. The algorithm starts by creating the initial partition P.

$$1, 5, 8, 9, 10 \mid 2, 3, 4, 6, 7$$

Once we reach Line 6 for the first time the internal state could be:

$$\begin{aligned} P &= 1, 5, 8, 9, 10 \mid 2, 3, 4, 6, 7 \\ C &= 1, 5, 8, 9, 10 \\ l &= a \\ P' &= 1, 5, 8, 9, 10 \mid 2, 3, 4, 6, 7 \end{aligned}$$

Note that the internal state could also be different, given that every $C \in P'$ and $l \in \Sigma$ need not be considered in any particular order.

On Line 6, the algorithm considers what classes are split with (C, l). Like mentioned above, it does so by creating a set I.

Consider the inverse δ-function as applied on every element of C with a.

$$\begin{aligned} \delta^{-1}(1, a) &= \emptyset \\ \delta^{-1}(5, a) &= \emptyset \\ \delta^{-1}(8, a) &= \{6, 8, 10\} \\ \delta^{-1}(9, a) &= \emptyset \\ \delta^{-1}(10, a) &= \emptyset \end{aligned}$$

The union of all of these sets is I, we can see that $I = \{6, 8, 10\}$. This means that the first class the first class is split as $1, 5, 9 \mid 8, 10$ since 8 and 10 are in I and $1, 5, 8$ are not. The second class is split as $2, 3, 4, 7 \mid 6$ such that $P = 1, 5, 8 \mid 8, 10 \mid 2, 3, 4, 7 \mid 6$.

The algorithm terminates when P and P' have not changed for an entire iteration of the "repeat until". The continuation of the algorithm can be seen in Table 3.1. The table shows the state as it is every time the algorithm reaches line 6. Iterations where C consists of a single state and no class is split are not shown.

P	C	l
$C' = 1, 5, 8, 9, 10 \mid 2, 3, 4, 6, 7$		
$1, 5, 8, 9, 10 \mid 2, 3, 4, 6, 7$	$1, 5, 8, 9, 10$	a
$1, 5, 9 \mid 8, 10 \mid 6 \mid 2, 3, 4, 7$	$1, 5, 8, 9, 10$	b
$1, 5 \mid 9 \mid 8, 10 \mid 6 \mid 2, 3, 7 \mid 4$	$2, 3, 4, 6, 7$	a
$1, 5 \mid 9 \mid 8, 10 \mid 6 \mid 2, 3, 7 \mid 4$	$2, 3, 4, 6, 7$	b

$C' = 1,5 \mid 9 \mid 8,10 \mid 6 \mid 2,3,7 \mid 4$		
$1,5 \mid 9 \mid 8,10 \mid 6 \mid 2,3,7 \mid 4$	$1,5$	a
$1,5 \mid 9 \mid 8,10 \mid 6 \mid 2,3,7 \mid 4$	$1,5$	b
$1,5 \mid 9 \mid 8,10 \mid 6 \mid 2,3 \mid 7 \mid 4$	$8,10$	a
$1,5 \mid 9 \mid 8,10 \mid 6 \mid 2,3 \mid 7 \mid 4$	$8,10$	b
$1,5 \mid 9 \mid 8,10 \mid 6 \mid 2,3 \mid 7 \mid 4$	6	a
$1,5 \mid 9 \mid 8,10 \mid 6 \mid 2 \mid 3 \mid 7 \mid 4$	6	b
$1,5 \mid 9 \mid 8,10 \mid 6 \mid 2 \mid 3 \mid 7 \mid 4$	$2,3,7$	a
$1,5 \mid 9 \mid 8,10 \mid 6 \mid 2 \mid 3 \mid 7 \mid 4$	$2,3,7$	b
$C' = 1,5 \mid 9 \mid 8,10 \mid 6 \mid 2 \mid 3 \mid 7 \mid 4$		
$1,5 \mid 9 \mid 8,10 \mid 6 \mid 2 \mid 3 \mid 7 \mid 4$	$1,5$	a
$1,5 \mid 9 \mid 8,10 \mid 6 \mid 2 \mid 3 \mid 7 \mid 4$	$1,5$	b
$1,5 \mid 9 \mid 8,10 \mid 6 \mid 2 \mid 3 \mid 7 \mid 4$	$8,10$	a
$1,5 \mid 9 \mid 8,10 \mid 6 \mid 2 \mid 3 \mid 7 \mid 4$	$8,10$	b
$1,5 \mid 9 \mid 8,10 \mid 6 \mid 2 \mid 3 \mid 7 \mid 4$	2	a
$1 \mid 5 \mid 9 \mid 8,10 \mid 6 \mid 2 \mid 3 \mid 7 \mid 4$	2	b
$C' = 1 \mid 5 \mid 9 \mid 8,10 \mid 6 \mid 2 \mid 3 \mid 7 \mid 4$		
$1 \mid 5 \mid 9 \mid 8,10 \mid 6 \mid 2 \mid 3 \mid 7 \mid 4$	$8,10$	a
$1 \mid 5 \mid 9 \mid 8,10 \mid 6 \mid 2 \mid 3 \mid 7 \mid 4$	$8,10$	b
$C' = 1 \mid 5 \mid 9 \mid 8,10 \mid 6 \mid 2 \mid 3 \mid 7 \mid 4$		

Table 3.1: Moore's algorithm as applied on Automata 3.1

3.2 Overview

Hopcroft first introduced his minimization algorithm in his 1971 paper *An n log n algorithm for minimizing states in a finite automaton* [7]. The title of this paper suggests that his algorithm has an $O(n \log n)$ complexity (with n the number of states). This is true, but only for a constant size alphabet. If we incorporate k for the size of the alphabet, we get $O(kn \log n)$, which is still a good improvement over the complexity of Moore's algorithm ($O(kn^2)$).

While Hopcroft does not reference Moore's algorithm, it can be said that Hopcroft's algorithm borrows many ideas from Moore. Mainly, choosing the initial partition, and refining until it represents the partition based on language equivalence. It improves on these ideas in several ways, but the

Algorithm 4 Hopcroft's Algorithm

```
Require: $A := (\Sigma, S, s_0, \delta, F)$
Ensure: The minimized automaton with the same language as A
 1: $P = \{F, S - F\}$
 2: $M = min(F, S - F)$ (the class with the least amount of states)
 3: $\mathcal{W} = \emptyset$
 4: for all $l \in \Sigma$ do
 5:     $\mathcal{W}.push((M, l))$
 6: end for
 7: while $\mathcal{W} \neq \emptyset$ do
 8:     $(S, l) = \mathcal{W}.pop()$
 9:     for all $C \in P$ for which $C$ is split by $(S, l)$ do
10:         $(C', C'') = c \mid (S, l)$
11:         Replace $C$ for $C'$ and $C''$ in $P$
12:         $C''' = min(C', C'')$
13:         for all $x \in \Sigma$ do
14:             for all $(C, x) \in \mathcal{W}$ (where $C$ is the same $C$ as above) do
15:                 Replace $(C, x)$ with $(C', x)$ and $(C'', x)$
16:             end for
17:             $\mathcal{W}.push((C''', x))$
18:         end for
19:     end for
20: end while
21: return a new DFA based on $A$ and $P$
```

most significant way is the *waiting set.* The waiting set is, in essence, a set of splitter that have yet to be considered for splitting classes in the partition. I will go more in detail, after the overview.

In this part, I first provide a global overview of the algorithm using the pseudocode in Algorithm 4, and continue to provide the reasoning I mentioned before.

Line 1 creates the initial partition.

Line 2 does preparational work for the loop on Line 4–6.

Line 3 initiates the waiting set. The waiting set is used to store a set of "splitters". Splitters are used to split the partitions that are in P at any given time.

Line 4–6 adds the smallest class to the waiting set.

Line 7 checks if the waiting set is empty. Since the splitting of a partition will always result in pushing to the waiting set (Line 17). It is trivial to realize that the algorithm must terminate. Every iteration one element is removed from the waiting set, and elements are only added when an partition is split. Thus, the waiting set will always reach an empty state once every partition is split.

Line 8 pops retrieves one element from the waiting set. This element is used in the splitting process on Line 9–18.

Line 9 loops over every class that is split by splitter (S, l).

Line 10-11 splits every class C that is split by (S, l), C is then replaced with the results of the split (c' and c'').

Line 12 selects the smallest of the two new classes to be added to the waiting set.

Line 14-16 Ensures that every class that was split in P is also split in the waiting set.

Line 17 adds only the smallest class to the waiting set.

3.3 Hopcroft's algorithm compared to Moore's algorithm

As mentioned before, Hopcroft's algorithm is very similar to that of Moore. However, there are some differences that decrease the complexity. I will now consider the differences with Moore's algorithm one by one.

Queue

Hopcroft realized that it is not needed to split based on the same splitter twice.

Consider the splitter (C, l) that is used to split some class C' into C'' and C'''. We will only consider C'' in the rest of this paragraph, but it trivially also holds for C'''. Since C' was split by (C, l), every state in C'' either has a transition to C, or none of the states do.

Since we consider every class that is split by (C, l), the resulting partition only has classes for which either all states transition to a state in C with l, or none. Splitting a class in a later stage will not compromise this property. As such, it is not needed to split on (C, l) again.

In order to incorporate this in the algorithm, Hopcroft uses some waiting set of splitters. What type of waiting set is used (Queue/Stack) is not specified in Hopcroft's 1971 paper. Research shows, however, that a FILO implementation is desired [3, 10].

The Smallest Class

Every time a class is split into two, the smallest of the two resulting classes is added to the waiting set. Since we must at some point use the δ^{-1}-function on every state in a splitter, using the smallest possibly splitters is preferred. It can be shown that only one of the two classes needs to be considered. Please note that the following considers only a single letter l, but that it holds for every letter in the alphabet.

Let C be a class that is split into C' and C''. there are two distinct cases: either (C, l) is in the waiting set, or it is not.

In the first case, both (C', l) and (C'', l) will end up in the waiting set due to the split. Given that the waiting set is indeed a set, adding one or the other will not change the contents of the set.

In the second case, another distinction must be made. Either it was in the waiting set at an earlier stage, or it was never in the waiting set. In the first case, we have already split based on (C, l). This, in turn, implies that for every class all states either have transitions that go to C via l, or none. When all states have such a transition, we can only split further based on whether states transition to C' or C'' with l. Since that is the only

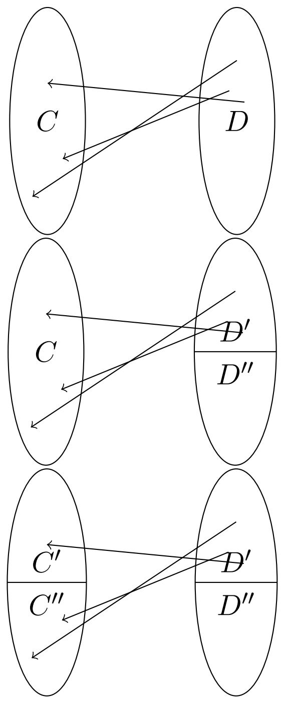


Figure 3.2: Smallest class

distinction that can be made, it is easy to realize that we can split on either (C', l) or (C'', l).

If no transitions go to C with l, there is nothing that can be gained from splitting on either C' or C''. So it does not matter which we add.

Consider Figure 3.2 where D is some class that is used to represent all other classes in the example above. The first figure shows the classes where we have not yet split C. The second figure shows the classes as it would be after splitting on (C, l), and the last figure shows the classes after C is split.

In the second case ((C, l) was never in the waiting set), we know that the its smaller counterpart C_2 (the smaller class that at some point formed a single class with this class) was added to the waiting set instead. We assume in the next paragraph that the order in which the splitters are considered does not impact the result of the algoritm. This is true, but not proven in this thesis.

We know that every class has only transitions to $C_2 \cup C$ with l, or none. Given the above we can assume that (C_2, l) is the the splitter that is considered first. (C_2, l) will split every class such that every class has only transitions to C_2 with l, C with l, or neither. Since we can only every split classes with transitions to C, it does not matter if we split on C' or C''.

3.4 Example

Let's consider Hopcroft's algorithm as it would be run on the automaton in Figure 3.1. The following table shows the inner state every time a splitter is popped from the waiting set.

0	P	$1, 5, 8, 9, 10 \mid 2, 3, 4, 6, 7$
	(S, l)	
	$\mathcal{W}$	$(1, 5, 8, 9, 10; a), (1, 5, 8, 9, 10; b)$
1	P	$1, 5, 9 \mid 8, 10 \mid 2, 3, 4, 7 \mid 6$
	(S, l)	$(1, 5, 8, 9, 10; a)$
	$\mathcal{W}$	$(1, 5, 9; b), (8, 10; b), (8, 10; a), (6; a), (6; b)$
2	P	$1, 5, 9 \mid 8, 10 \mid 2, 3, 7 \mid 4 \mid 6$
	(S, l)	$(1, 5, 9; b)$
	$\mathcal{W}$	$(8, 10; b), (8, 10; a), (6; a), (6; b), (4; a), (4; b)$
3	P	$1, 5 \mid 9 \mid 8, 10 \mid 2, 3, 7 \mid 4 \mid 6$
	(S, l)	$(8, 10; b)$
	$\mathcal{W}$	$(8, 10; a), (6; a), (6; b), (4; a), (4; b), (9; a), (9; b)$
4	P	$1, 5 \mid 9 \mid 8, 10 \mid 2, 3, 7 \mid 4 \mid 6$
	(S, l)	$(8, 10; a)$
	$\mathcal{W}$	$(6; a), (6; b), (4; a), (4; b), (9; a), (9; b)$
5	P	$1, 5 \mid 9 \mid 8, 10 \mid 2, 7 \mid 3 \mid 4 \mid 6$
	(S, l)	$(6; a)$
	$\mathcal{W}$	$(6; b), (4; a), (4; b), (9; a), (9; b), (3; a), (3; b)$
6	P	$1, 5 \mid 9 \mid 8, 10 \mid 2, 7 \mid 3 \mid 4 \mid 6$
	(S, l)	$(6; b)$
	$\mathcal{W}$	$(4; a), (4; b), (9; a), (9; b), (3; a), (3; b)$
7	P	$1, 5 \mid 9 \mid 8, 10 \mid 2, 7 \mid 3 \mid 4 \mid 6$
	(S, l)	$(4; a)$
	$\mathcal{W}$	$(4; b), (9; a), (9; b), (3; a), (3; b)$
8	P	$1, 5 \mid 9 \mid 8, 10 \mid 2, 7 \mid 3 \mid 4 \mid 6$
	(S, l)	$(4; b)$
	$\mathcal{W}$	$(9; a), (9; b), (3; a), (3; b)$
8	P	$1, 5 \mid 9 \mid 8, 10 \mid 2, 7 \mid 3 \mid 4 \mid 6$
	(S, l)	$(9; a)$

	$\mathcal{W}$	$(9;b),(3;a),(3;b)$
9	P	$1,5 \mid 9 \mid 8,10 \mid 2 \mid 7 \mid 3 \mid 4 \mid 6$
	(S,l)	$(9;b)$
	$\mathcal{W}$	$(3;a),(3;b),(2;a),(2;b)$
10	P	$1,5 \mid 9 \mid 8,10 \mid 2 \mid 7 \mid 3 \mid 4 \mid 6$
	(S,l)	$(3;a)$
	$\mathcal{W}$	$(3;b),(2;a),(2;b)$
11	P	$1,5 \mid 9 \mid 8,10 \mid 2 \mid 7 \mid 3 \mid 4 \mid 6$
	(S,l)	$(3;b)$
	$\mathcal{W}$	$(2;a),(2;b)$
12	P	$1 \mid 5 \mid 9 \mid 8,10 \mid 2 \mid 7 \mid 3 \mid 4 \mid 6$
	(S,l)	$(2;a)$
	$\mathcal{W}$	$(2;b),(1;a),(1;b)$
13	P	$1 \mid 5 \mid 9 \mid 8,10 \mid 2 \mid 7 \mid 3 \mid 4 \mid 6$
	(S,l)	$(2;b)$
	$\mathcal{W}$	$(1;a),(1;b)$
14	P	$1 \mid 5 \mid 9 \mid 8,10 \mid 2 \mid 7 \mid 3 \mid 4 \mid 6$
	(S,l)	$(1;a)$
	$\mathcal{W}$	$(1;b)$
15	P	$1 \mid 5 \mid 9 \mid 8,10 \mid 2 \mid 7 \mid 3 \mid 4 \mid 6$
	(S,l)	$(1;b)$
	$\mathcal{W}$	$(1;b)$

Table 3.2: Hopcroft as run on the automaton in Figure 3.1

3.5 Implementation

The algorithms in this thesis were implemented in C++. This section will highlight key parts of the algorithm, and consider their implementation in C++.

δ^{-1} over δ

Line 9 of Hopcroft's algorithm requires determining which of the classes are split by the splitter. In order to do this in the least amount of time possible, we make use of the inverse delta function: $\delta^{-1} : S \times \Sigma \to \mathcal{P}(S)$. This function takes some state and letter, and returns the states that reach that state with that letter, as defined in the preliminaries. In order to allow this function to be performed in $O(1)$, the inverse transition table is calculated before Line 9 is reached.

```cpp
for (DFAState* state : dfa->states) {
  state->reverse_transitions.resize(state->transitions.size());
}
for (DFAState* state : dfa->states) {
  for (unsigned int i = 0; i < state->transitions.size(); i++) {
    state->transitions[i]->reverse_transitions[i].push_back(
    state);
  }
}
```

Class to Splitter map

An optimal implementation of line 14 in the algorithm requires an $O(1)$ mapping between a class and the splitters that are based on this partition.

```cpp
class Class {
  public:
    std::list<DFAState*> states;

    ⋮
    std::list<Splitter*> splitters;
};
```

Besides classes knowing all splitters that are based on it, the opposite is also true.

```cpp
class Splitter {
  public:
    Class* c;
    unsigned int letter;
};
```

The class in the splitters enables Line 15 to be optimally implemented. There is no need to split class and splitters separately. Splitting the class means that the splitter is also split. Of course, the other resulting class still needs to be added to $\mathcal{W}$.

Achieving high performance on Line 9 is done in a similar manner. Every class consists of a set of states, every state has a pointer to the class to which is belongs. The combination of this mechanic enables us to consider only class that could potentially be split, ignoring the others.

```cpp
for (DFAState* s : splitter->p->states) {
  for (DFAState* orig_state : s->reverse_transitions[splitter->
    letter]) {
    if (orig_state && !(orig_state->different_part)) {
      orig_state->different_part = true;
      Class* temp = orig_state->p;
      if (temp->times_added == 0) {
        result.push_back(temp);
      }
      temp->times_added++;
    }
  }
}
```

The times_added variable is later used to determine if every every/no state had transitions to the splitter, if this is case, the class as a whole is not split.

3.6 Complexity of Implementation

The focus of this paper lies not with a formal analysis of the complexity of the implementation. Instead, I would like to refer to an unpublished paper by Hang Zhou on implementing Hopcroft's algorithm [13]. Which describes that datastructures that are needed for an optimal implementation.

Chapter 4

Brzozowski's Algorithm

4.1 Overview

Brzozowski's algorithm is a conceptually easy algorithm that was first described by Brzozowski in 1962 [5] and has an exponential time and space worst case complexity. Pseudocode of the algorithm can be found in Algorithm 5. Unlike Moore's and Hopcroft's algorithms, Brzozowski's algorithm can also be applied to NFAs. Given the focus of this thesis, however, the rest of the chapter will only consider DFAs.

Algorithm 5 Brzozowski's Algorithm

```
Require: $DFA := (\Sigma, S, s_0, \delta, F)$
Ensure: $DFA := (\Sigma, S', s'_0, \delta', F')$
1: $det(rev(det(rev(DFA))))$
```

Where "det" and "rev" are the determinization and reversal functions that were defined in the preliminaries.

4.2 Example

In order to understand the workings of the algorithm, this section provides an example for the algorithm. Consider the non-minimal automaton in Figure 4.1, Brzozowski's algorithm starts by reversing the automaton. Figure 4.2 shows the reversal of this automaton.

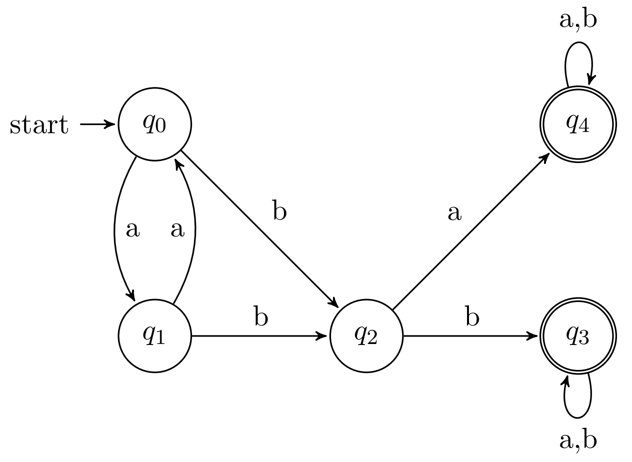


Figure 4.1: Complete Automaton 2

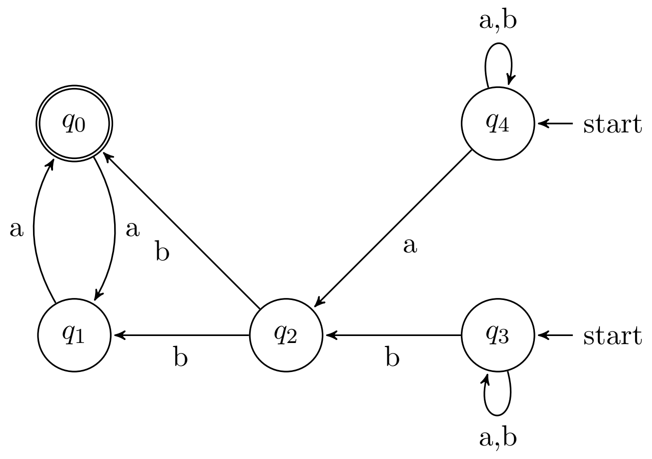


Figure 4.2: Reversed Automaton 1

Subsequently, the resulting NFA is converted to a DFA using the subset construction method described in Algorithm 1. The resulting DFA can be seen in Figure 4.3

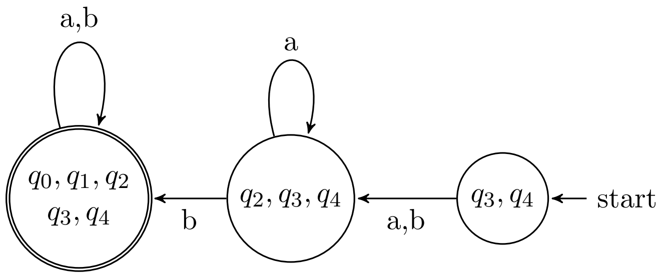


Figure 4.3: Determinized Automaton 1

This procedure (reversal and determinization) is performed once more, to create the automata in Figure 4.4.

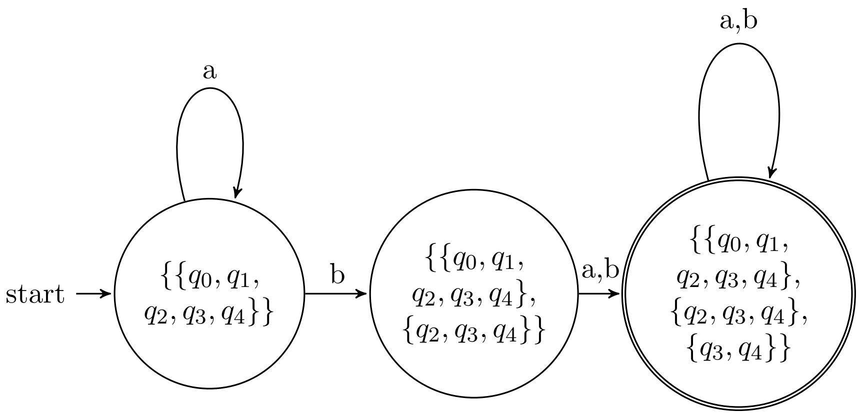


Figure 4.4: Resulting DFA from Brzozowski's Algorithm

4.3 Implementation

The implementation of Brzozowski's algorithm consists of two main parts, reversal and subset construction. I will consider both individually.

Reversal

Unlike the pseudocode in Algorithm 2, the code does not consider the final states seperately from the reversing of the edges. Rather, every transition is considered, and special actions are performed if the originating DFA state is an initial state. After that, the new transition is added to the NFA. Since DFAs and NFAs use a different data structure, NFA states are constructed at runtime everytime a NFA state is encountered that does not exist yet.

```
for (DFAState* d_state : states) {
  for (unsigned int i = 0; i < d_state->transitions.size(); i++)
    {
```

```
    ⋮
    n_state_from = result->states.at(d_state->id);
    ⋮
    n_state_to = result->states.at(d_state->transitions.at(i)->
   id);
    ⋮
    n_state_from->transitions.push_back(std::make_pair(
   n_state_to, i));
  }
}
```

Subset Construction

The Implementation of subset construction is a translation of the pseudocode in Algorithm 1. There are, however, multiple ways to implement this algorithm. Subset construction, when implemented with an optimal time complexity, has a space complexity of $\Theta(2^n)$. Experimental results show that such an implementation is not desired in practice due to the large nature of the automata. An alternate implementation exists that trades some time efficiency for a $O(2^n)$ space complexity. This section first describes parts of the implementation that are not linked to this difference, and will then compare the two versions.

In order to implement this algorithm with with a minimal time complexity, we must ensure that the statements on Line 4 and 5 can be performed in minimal time.

If we consider Line 4 without any context, one might be tempted to say that this statement can be performed in $O(1)$. However, since we must also collect all destinations in some accumulator, the implementation will still be $O(|E|)$. See the Further Improvements section for a more detailed reasoning.

In order to implement Line 4 with such a complexity, we need an $O(1)$ mapping from a DFA state to the set of NFA states that it represents. In the code, I use a vector to achieve this:

```
std::vector<std::vector<NFAState*>*>* d_to_n =
    new std::vector<std::vector<NFAState*>*>;
```

Whenever a new DFA is constructed during the algorithm, it is assigned an identifying integer. This integer is also used as the index for the set of NFA states in the vector. One might argue that it would have been a better idea to make the set of NFA states a property of the DFA state, but I chose not to implement it in an attempt to keep the code more understandable.

Line 5 seems, again, implementable in $O(1)$. This assumption rests on the idea that, since we already have all NFA states accumulated, we need but add the set to the d_to_n vector and the queue. We must, however, also consider that the DFA state might already exist, if we do not there are cases in which the algorithm might not terminate. I have implemented two ways to achieve this. The first is theoretically the fastest, but has a $\Theta(2^n)$ complexity. The other is a little slower, but has a $O(2^n)$ complexity. I will no consider each of these ways.

Version 1 ($\Theta(2^n)$) Conceptually, we need to solve this issue by being able to access a DFA state given a set of NFA states in $O(1)$. This seems impossible, since it would take at least $O(n)$ (with n the number of NFA states in the set) to determine uniqueness. We can, however, leech on the steps performed by the accumulation of NFA states.

My specific implementation considers some set of NFA states as a bitvector $(s_n, s_{n-1}, \dots, s_1, s_0)$ where s_i is the NFA state with id i. This bitvector is, in turn, considered as a integer and used as an index in a vector:

```
1 std::vector<DFAState*>* n_to_d =
2     new std::vector<DFAState*>(pow(2, states.size()));
```

The vector is created with size $2^{|S|}$ in order to fix the maximum integer that could result from the bitvector.

Relevant parts of the code that mentioned above are:

```
1 for (auto initial_state : initial_states) {
2   index |= (ONE << initial_state->id);
3 }
```

Which is where the index is constructed to map a set of NFA states to a DFA state for the initial states and:

```
1 for (auto n_state_c : n_state->move(letter)) {
2   if (!n_state_c->added_to_acc) {
3     index |= (ONE << n_state_c->id);
4     accumulator->push_back(n_state_c);
      ⋮
5   }
6 }
```

where the same is done for all other states. The actual lookup in the n_to_d vector is done in:

```
1 DFAState* target = n_to_d->at(index);
2 if (target == nullptr) {
3   target = new DFAState(DFA_id_counter, alphabet->size());
4   DFA_id_counter++;
5   d_to_n->push_back(accumulator);
6   n_to_d->at(index) = target;
      ⋮
7 }
```

Version 2 ($O(2^n)$) If we choose not to implement the step on Line 5 optimally (by using a std::map), we can reduce the space complexity to

$O(2^n)$ at the cost of some performance. The idea is relatively simple; we replace the n_to_d vector with a map. This allows us not to have to initialize the data structure, which in turn means that we do not have to reserve a 2^n size memory block. Version 2 is the version that is used for the benchmarks in this thesis.

4.4 Further Improvements

My implementation of Brzozowski's algorithm can still be improved on some fronts to increase the efficiency. In this section I will consider some of the improvements that I considered implementing, but were left out due to time constraints.

Reversal

In order to provide context to the changes I suggest, important parts of the code are provided first.

```
NFAState* n_state_from;
NFAState* n_state_to;
for (DFAState* d_state : states) {
  for (unsigned int i = 0; i < d_state->transitions.size(); i++)
    {
    if (result->states.at(d_state->id) == nullptr) {
      n_state_from = new NFAState(d_state->id);
      result->states.at(d_state->id) = n_state_from;
    } else {
      n_state_from = result->states.at(d_state->id);
    }
    if (result->states.at(d_state->transitions[i]->id) ==
   nullptr) {
      n_state_to = new NFAState(d_state->transitions[i]->id);
      result->states.at(d_state->transitions[i]->id) =
   n_state_to;
    } else {
      n_state_to = result->states.at(d_state->transitions[i]->id
   );
    }
    n_state_to->transitions.push_back(std::make_pair(
   n_state_from, i));
  }
}
```

Currently, there are several parts of the code that need not be in the inner loop. Since we go through every transition of a given state, we know that n_state_to will stay the same in a single iteration of the outer for-loop. This means that we could move the check for a nullptr, and the initial/accepting state check one for-loop up. We can do even better by considering the accepting state outside the outer for-loop. This can easily be done by using the fact that the DFA data structure contains a vector with pointers to the accepting states.

Subset Construction

Currently, vectors are used to store the NFA states to DFA state mapping. Since the size is known during the initialization of the vector, using an array could significantly reduce the overhead.

In Section 4.3 I discussed the different data structures that can be used to store the n_to_d mapping. There is another option that lays outside the scope of this thesis, and has the same complexity as a map. Despite the same complexity as the map, in practice, it might perform differently from std::map. The concept of the new data structure is that it is a tree that is traversed as the transitions are considered. In theory, this means that the tree has been fully traversed after the inner forloop. One difficulty with this implementation is that every transition must be considered concurrently and in order. Forcing the incremental order of the transitions can be done in construction. If we then use a merging algorithm to merge all transition vectors, we can get a single vector containing all destination states. Merging two vectors of length n and m van be done in $O(m + n)$. As such, access and insertion can be done in the same complexity as in a std::map.

Chapter 5

Experiments

5.1 Code

The implementations used in the experiments are written in C++14. They use the aforementioned techniques to create the most optimal versions of the algorithms. The implementations, including the Random Automata Generator written in Clean can be found in the git repository [1].

5.2 Benchmark Set

A benchmark is needed to determine the performance of the algorithms. In order to ensure that their practical performance is tested, it is needed that the automata in the benchmark are from practical applications. Random automata might not have similar characteristics to the automata that are used in practice. For this reason, a benchmark set originating from model checking is used (see subsection 5.2.1). In order to provide contrast, and address a possible issue regarding a characteristic of the automata, random automata were also generated (see subsection 5.2.2).

5.2.1 Model Checking Automata

The automata used in this thesis were generated from the intermediate steps of model checking [1], with the purpose of benchmarking.

The paper for which they were created is not the only paper in which they were used in benchmarking. In particular, they have been used to benchmark an optimization of the algorithm by Hopcroft and Karp, using a technique called *bisimulation up to congruence* [4]. In total, there are 1604 automata in the benchmark set.

The set has been incorporated into the "automatark"[2] benchmark set,

[1]gitlab.science.ru.nl/eveen/Bachelor-Thesis
[2]github.com/lorisdanto/automatark

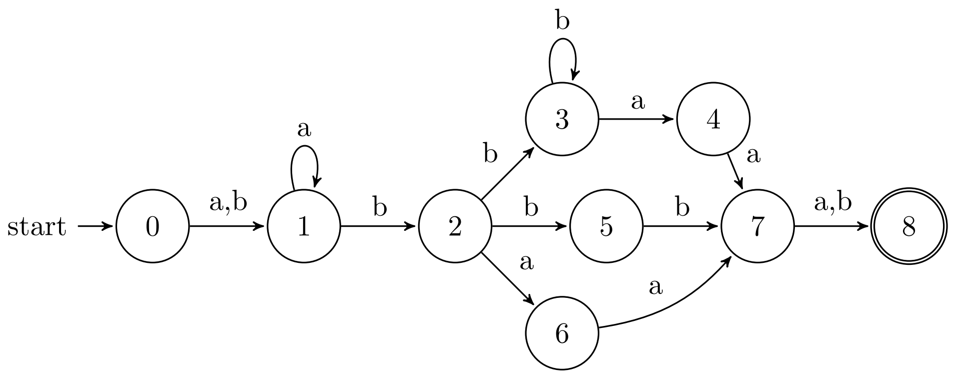


Figure 5.1: Benchmark Automaton 1

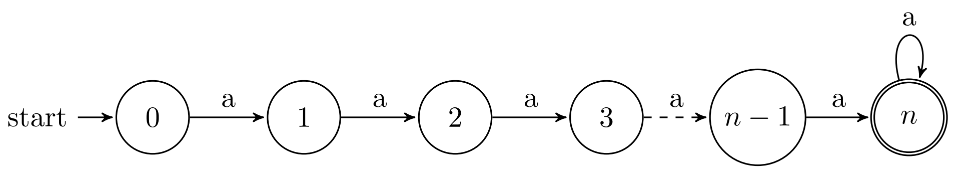


Figure 5.2: Brzozowski's algorithm Worst Case

which contains several types of automata that were all aggregated with the purpose of benchmarking.

Due to the origin of the automata, they have some characteristics that might not be ideal in the case of comparing the performance of the algorithms implemented in this thesis. In particular, one such characteristic is that they have no cycles that contain more than one state. That is, all automata look somewhat like the automaton in Figure 5.1.

The fact that the automata have no cycles consisting of more than 1 state could negatively influence the balance of the benchmark, especially considering the similarity of these automata compared to Brzozowski's algorithm's worst case automaton shown in Figure 5.2.

Additionally, all automata are NFAs rather than DFAs (that would both work with Hopcroft's algorithm and Brzozowski's algorithm). In order to more accurately benchmark the algorithms, all NFAs are first converted to DFAs.

5.2.2 Random Automata

In order to allow for a more balanced benchmark, random automata were generated using a modified version (in Clean) of the generator used by Bonchi and Pous [4]. I used the following variables to generate the automata: a linear transition density of 1.25 (every state has an expected out-degree of 1.25 for every letter) as proposed by Tabakov and Vardi [11];

a 0.1 chance of a state being an initial state; and a 0.1 chance of state being an accepting state. State 1, is always both an initial state, and an accepting state. The following code is used to generate the automata:

```
 1 random_nfa :: Int Int Real Real Real [Real] -> NFA
 2 random_nfa n v r pa pi randReals
 3 # d = r / (toReal n)
 4 # randDeltas = diag3 [1 .. n] [1 .. v] [1 .. n]
 5 # randDeltas = dropOnChance randDeltas randReals d
 6 # randReals = drop (n * n * v) randReals
 7 # accepting = [1] ++ (dropOnChance [2 .. n] randReals pa)
 8 # randReals = drop n randReals
 9 # initial = [1] ++ (dropOnChance [2 .. n] randReals pi)
10 = { size = [1 .. n],
11   delta = randDeltas,
12   accept = accepting,
13   initial = initial}
```

Where `n` is the amount of states in the final automaton, `v` is the amount of letters in the alphabet, `r` is the transition density, `pa` is the probability of a state being an accepting state, `pi` is the probability of a state being an initial state and `randReals` is an infinite list of random Reals between 0 and 1 that have been read from `/dev/urandom`.

Tranition density

In order to calculate the probability that a transition should not be removed from the list of all possible transitions, a translation must be made from the transition density to the aforementioned probability. This is simply done using the expected value.

Assuming that every transition has an equal probability, we want the expected value to be: $r \cdot n \cdot v$. If we then use the formula for the expected value, we get:

$$r \cdot n \cdot v = n \cdot n \cdot v \cdot p$$

where p is the probability that we want. Rewriting the formula gives:

$$p = \frac{r}{n}$$

As such, that is how the probability is calculated on line 3.

This code is used to generate 1820 automata, 20 for every number of states between 10 and 100.

5.3 Integrity

In order to ensure the integrity of the benchmark, several measures were implemented. I will go over all factors that I considered could have an influence on the integrity, and explain how I tried to avoid them.

Caching Caching is a hardware implemented behaviour by the CPU that allows some part of RAM to be stored in small temporary storage closer to the CPU. This enables faster access to parts of code and constants that are used often. It is possible that either Hopcroft's algorithm or Brzozowski's algorithm benefits more from this than the other. As such, the program, and all it's data structures, are removed from memory upon completion of the minimization of an automaton. Note that this does not completely disable caching. Rather, it reduces that advantage and either algorithm might have over the other.

Parsing Two I/O operations on the same data in a computer can take a different time. I think this difference is enough to influence the result. As such, the reading (and parsing) of the automata is not part of the time that is counted for a run of either of the algorithms.

Auto Clocking/Turbo Boost The motherboard and CPU used in the benchmarking machine support auto clocking and turbo boost. These technologies are designed to reduce the power use and temporarily increase the power of a PC respectively. If the clock speed is reduced or increased during certain parts of the benchmark, this could be in favor of either of the algorithms. Therefore, both Auto Clocking and Turbo Boost were disabled for the benchmark.

Others In order to further improve the integrity, the benchmark was run on a single day on a single machine. The machine was not used during the run of the benchmark, and all non-essential processes were killed before the benchmark was run.

Specs The specs of the machine were:

OS Arch Linux x64

CPU Intel Core i5-4460 @ 3.20GHz

RAM 8GB @ 1333 MHz

Auto Clocking/Turbo Boost Disabled

5.4 Results

For brevity, all plots will show Brzozowski's algorithm as "BRZ" and Hopcroft's algorithm as "HOP". Every run of an algorithm on a given automata was given a 5 minute time limit. If the algorithm did not terminate in that time, the data was not included in the result. There were 46 instances where this occurred. In total, 8064 tests were done.

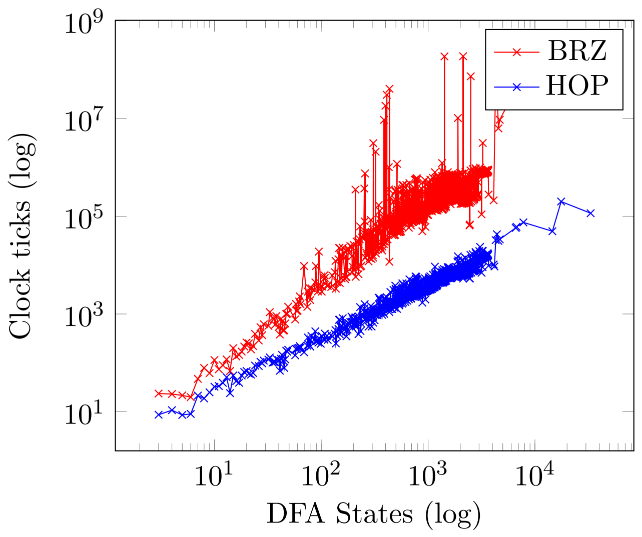


Figure 5.3: The algorithms when tested on the benchmarkset. The y-axis shows the clock ticks that the benchmark machine needed to calculate the minimal DFA. The x-axis shows the number of states that were in the original DFA (after the initial conversion from a NFA).

5.4.1 The Benchmark Set

The algorithms were first run on every automaton in the benchmark set. Figure 5.3 shows the *clock ticks* that a specific run of an algorithm took.

Tabakov and Vardi [11] showed that the relative performance of the algorithms changes as the transition density of the orignal NFA changes. In order to test this on the benchmark set, the transitions densities were calculated and rounded to the nearest tenth. The results can be found in Figure 5.4. The graph shows the ratio of the time BRZ took for the automata and the time HOP took for the automata. A number below 1 means that HOP took less time to minimize the automata than BRZ.

5.4.2 The Random Automata

Next, the algorithms were run on the randomly generated automata. The results of these experiments can be found in Figure 5.5.

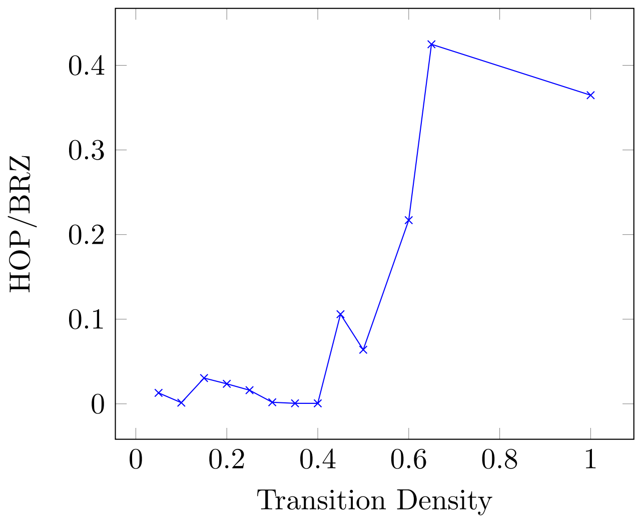


Figure 5.4: The algorithms when tested on the benchmarkset. The x-axis shows the transition density, the y-axis shows the ratio of the clock ticks. In particular, it shows the average clock ticks HOP needed for the automata divided by the average clock ticks BRZ needed for the automata.

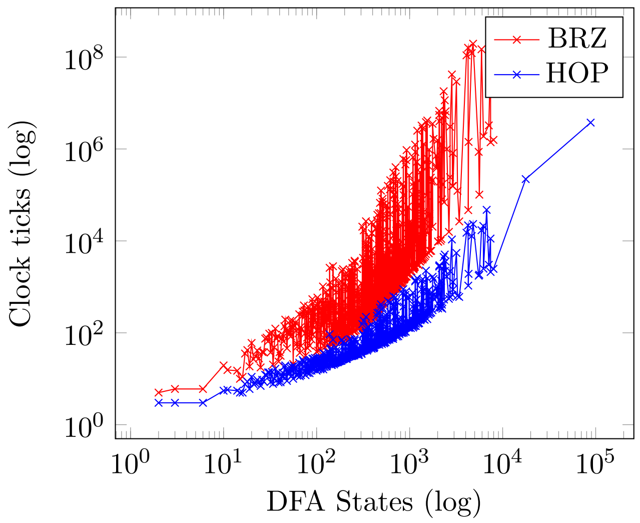


Figure 5.5: The algorithms when tested on the random automata. The y-axis shows the clock ticks that the benchmark machine needed to calculate the minimal DFA. The x-axis shows the number of states that were in the original DFA (after the initial conversion from a NFA).

Chapter 6

Related Work/Discussion

There has been some research into the practical performance of automata minimization algorithms, most notably the work by Watson [12], Tabakov and Vardi [11], and Almeida et al [2]. I will consider each of these works in chronological order. For each of the works I will explain the reasoning behind their research as well as the results that they obtained, followed by a discussion regarding the results compared to those obtained in this thesis.

6.1 Watson, 1995

Watson indicated that is is hard to compare algorithms that achieve the same result because of several factors.

1. Different programming languages
2. No implementations
3. No collective framework

All of these boil down to a single, comprehensive problem, there is no set of implemented algorithms that are part of the same framework. He argues that this makes it impossible to correctly compare the performance of these algorithms. As such, in his thesis, Watson provides such a framework and compares the running time of the algorithms he implemented.

The result of his experiments can be found in Figure 6.1.

BRZ is Brzozowski's algorithm, HOP is Hopcroft's algorithm, and BW is an algorithm due to Watson himself. Noteworthy is the fact that Brzozowski's algorithm perform consistently better than Hopcroft's algorithm, and that Watson's algorithm performs better than Brzozowski's algorithm in most cases. This is exceptionally unintuitive when considering their theoretical complexity.

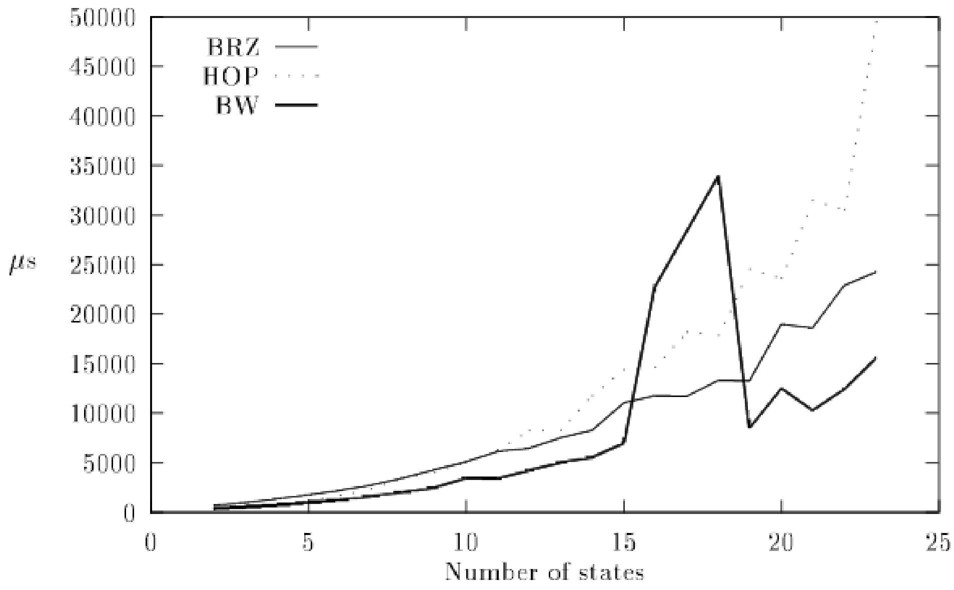


Figure 6.1: Median performance versus DFA states by Watson (lower is better)

The DFAs that Watson used in his research were obtained from random regular expressions or they were generated from "some other specification" [12].

When comparing these results to the ones obtained in this thesis, Hopcroft's algorithm appears to perform faster than Brozowski's algorithm, instead of the other way around. This can be clearly seen in Figure 5.3 and 5.5.

6.2 Tabakov and Vardi, 2005

Tabakov and Vardi tested the algorithms on randomly generated automata. In particular, they devote a relatively large section of their paper to determining the best way to generate such automata. It is this study that I use to generate the random automata in the thesis.

Their results show that Brzozowski performs significantly better when the automata to minimize has a high (> 1.5) transition density.

While this doesn't go completely against the results that were found by Watson, it does show that Brzozowski's algorithm does not perform strictly better than Hopcroft's algorithm.

When comparing Figure 5.4 with those provided by Tabakov and Vardi [11], it is clear that the benchmark set does not have a wide enough range of transition densities to draw a significant conclusion. One can see, however, that the ratio increases as the transition density increases. Unfortunately, this

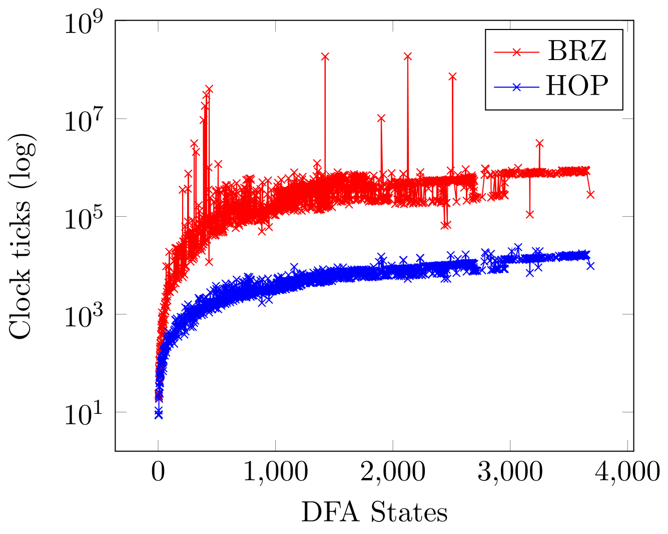


Figure 6.2: Figure 5.3 when the last few results are removed and the x-axis is not logarithmic.

ratio cannot be compared to the data as presented by Tabakov and Vardi, because they do not provide good contrast in the lower ranges of the transition density.

With regards to the number of states, Tabakov and Vardi noted that "At the peak, Hopcroft's algorithm scales exponentially, but generally the algorithms scale subexponentially" [11]. When we make slight modifications to the representation of the results in Figure 5.3, similar conclusions can be drawn from my experiments (see Figure 6.2).

Comparing the results that I obtained (see Figure 6.3) on the random automata, with the results obtained by Tabakov and Vardi for a density of 1.25, shows similar growth as well.

6.3 Almeida et al, 2007

Almeida et al. argue that the automata used by Watson are biased, and suggest using random automata to assess the performance of the algorithms. Their approach to generating automata is different from the one used by both Watson (using regular expressions), and Tabakov and Vardi (using their own generator). Instead, they opt to represent automata as a string, and then randomly generating these strings. They also argue, however, that this method provides uniform random samples, and thus does not represent typical use of the algorithms.

Their results show that Brzozowski's algorithm (as well as Watson's algorithm) does not perform well in any case, and that an improved version

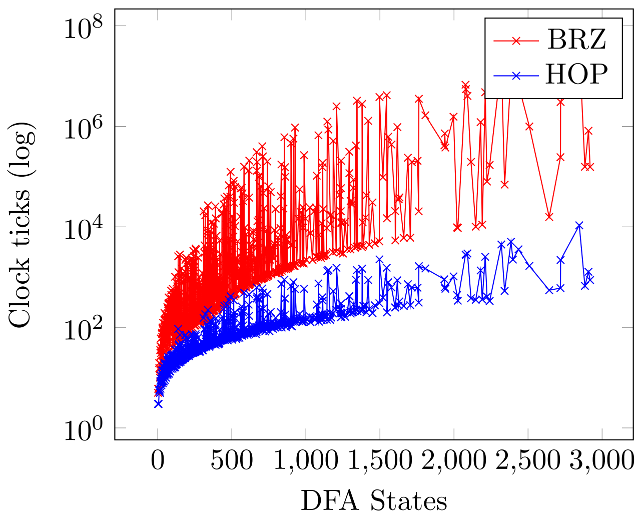


Figure 6.3: Figure 5.5, but with the last few results removed, and the x-axis no longer logarithmic.

of Watson's algorithm (with memoization) performs better than Hopcroft's algorithm for automata with more than 10 states. In their experiments on pseudo randomly generated NFAs, they show that Hopcroft's algorithm and Brzozowski's algorithm perform nearly identical for a variety of transition densities. Unfortunately, a comparison with the results obtained by Tabakov and Vardi is hard, due to the small range of transition densities that were tested by Almeida et al.

Almeida et al. show that Brzozowski's algorithm is largely outperformed by Hopcroft's algorithm when used on the equally distributed DFAs [2]. While this is more in line with the results obtained in this thesis, the time limit that was opposed by Almeida et al. leads to incomplete data on the performance of Brzozowski's algorithm. Therefore, we can only compare the results that were obtained for automata with 10 states. For these automata, the results of the two experiments seem very similar, with Hopcroft's algorithm performing approximately a factor 10 better than Brzozowski's algorithm.

Almeida et al. also provide results of experiments where transition densities were compared. Their experiments include sets with a transition density of 0.2, 0.5 and 0.8, with an alphabet size of 2 and 5 letters. The results that they obtained show virtually identical performance between the two algorithms, even when subset construction is included in the timing of the algorithms. As mentioned before, it is hard to compare these results to those of Tabakov and Vardi. It is, however, interesting to note that these results do not resemble the results that were obtained in this thesis. This discrepancy is likely due to the fact that this thesis didn't use randomly generated automata to get the data.

6.4 Conclusion

In conclusion, this thesis shows that Hopcroft's algorithm outperforms Brzozowski's algorithm in the benchmark set. However, due to the relatively small transition densities of the automata in the benchmark set, a strict conclusion cannot be drawn in the general case. If the assumption is made that the benchmark set fairly represents the automata used in practice, Hopcroft's algorithm is shown to perform better in practice.

6.5 Further Work

The main issue with the benchmark set used in this thesis is that it might not accurately represent the automata that are used in practice. In order to ensure that this is the case, it might be a sensible idea to collect automata that are actually used in practice. I propose that regular expressions are collected from source code files in public repositories (e.g. GitHub) and are subsequently converted to automata to form a new benchmark set.

Although not for minimization, the fact that these automata are from source code repositories ensures that they are at least used in practice.

Bibliography

[1] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. When simulation meets antichains. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 158–174. Springer, 2010.

[2] Marco Almeida, Nelma Moreira, and Rogério Reis. On the performance of automata minimization algorithms. *Logic and Theory of Algorithms*, page 3, 2007.

[3] Manuel Baclet and Claire Pagetti. Around hopcroft's algorithm. In *International Conference on Implementation and Application of Automata*, pages 114–125. Springer, 2006.

[4] Filippo Bonchi and Damien Pous. Checking nfa equivalence with bisimulations up to congruence. In *ACM SIGPLAN Notices*, volume 48, pages 457–468. ACM, 2013.

[5] J.A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. *Mathematical Theory of Automata*, 12:529–561, 1962.

[6] Pedro García Gómez, Damián López Rodríguez, and Manuel Vázquez-de-Parga Andrade. Dfa minimization: from brzozowski to hopcroft. 2013.

[7] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, DTIC Document, 1971.

[8] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *ACM SIGACT News*, 32(1):60–65, 2001.

[9] Edward F Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[10] Andrei Păun, Mihaela Păun, and Alfonso Rodríguez-Patón. Hopcroft's minimization technique: Queues or stacks? In *International Confer-*

ence on Implementation and Application of Automata, pages 78–91. Springer, 2008.

[11] Deian Tabakov and Moshe Y. Vardi. Experimental evaluation of classical automata constructions. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 396–411. Springer, 2005.

[12] Bruce William Watson et al. *Taxonomies and toolkits of regular language algorithms*. Eindhoven University of Technology, Department of Mathematics and Computing Science, 1995.

[13] Hang Zhou. Implementation of the hopcroft's algorithm. 2009.