UNIVERSITÄT
BAYREUTH

Fakultät für Mathematik, Physik und Informatik
Institut für Informatik
Lehrstuhl für Angewandte Informatik 7
Theoretische Informatik

Bachelorarbeit

Generation of DFA Minimization Problems

Generierung von DFA Minimierungsproblemen

Gregor Hans Christian Sönnichsen

February 22, 2020

Betreuer:
Prof. Dr. Wim Martens
M.Sc. Tina Trautner

Prüfer:
Prof. Dr. Wim Martens
?

Abstract

The theory of deterministic finite automatons (DFAs) is a classical topic of computer science-related courses. A typical task for students is to minimize a DFA. However generation of those DFAs that shall be minimized is often done manually by the exercise instructor. This work presents ideas to automize the generation of DFA minimization tasks.

We start in chapter 1 with introducing minimization tasks, which consist of a DFA A_{task} which has to be minimized and the minimal solution DFA A_{sol}. We focus on the minimization algorithm by Hopcroft, which works in two steps: Firstly delete unreachable states, then merge equivalent state pairs.

Following this separation in reverse, our approach is to generate the solution DFA first, then create equivalent state pairs and lastly add unreachable states. We devise several sensible input parameters and requirements for each of these stages.

Concerning the generation of solution DFAs (chapter 2) we make use of a simple rejection algorithm, that generates test DFAs by randomization or enumeration. Test DFAs are rejected, if they do not match the demanded properties. On this topic research has already been active, an overview about results there is made to draw conclusions for this work.

In chapter 3 we describe the extension of solution DFAs towards a task DFA. To archive this, we can add states and transitions in an easy manner according to certain rules, which are derived from the properties equivalent state pairs and unreachable states.

Zusammenfassung

Automatentheorie ist ein klassisches Thema in Lehre mit Informatikbezug. Eine typische Aufgabe für Studenten ist die Minimierung eines deterministischen endlichen Automaten (DEAs). Das Generieren solcher Minimierungsaufgaben wird allerdings häufig manuell vom Übungsleiter vorgenommen. In dieser Arbeit werden somit Ideen präsentiert um DEAs automatisiert zu generieren.

Wir beginnen in Kapitel 2 mit einer Beschreibung von Minimierungsaufgaben, die im Wesentlichen aus einem *Aufgaben-DEA* A_{task}, dem zu minimierenden DEA, und dem bereits minimierten *Lösungs-DEA* A_{sol} bestehen. Wir werden uns hier auf den Minimierungsalgorithmus von Hopcroft beschränken, der in zwei Schritten abläuft: Zunächst werden unerreichbare Zustände entfernt und dann äquivalente Zustandspaare zusammengefasst.

In unserem Ansatz nutzen wir diese Zweiteilung indem wir sie umdrehen, sodass zunächst der Lösungs-DEA generiert wird, woraufhin äquivalente Zustandspaare erzeugt und unerreichbare Zustände hinzugefügt werden. Für jeden dieser Schritte werden wir diverse sinnvolle Eingabeparameter und Anforderungen definieren.

Um die Lösungs-DEAs zu generieren (Kapitel 3) machen wir Gebrauch von einem simplen Algorithmus, der wiederholt Test-DEAs mittels Randomisierung oder Enumerierung erzeugt und sie immer dann ablehnt, wenn sie den gewünschten Eigenschaften nicht entsprechen. Zu diesem Thema gab es bereits einige Forschungsarbeit, folglich werden wir einen Überblick über relevante Ergebnisse geben um dann Schlussfolgerungen für diese Arbeit zu ziehen.

In Kapitel 4 beschreiben wir, wie Lösungs-DEAs zu Aufgaben-DEAs erweitert werden können. Um das zu erreichen können wir Zustände und Transitionen recht einfach mithilfe gewisser Regeln hinzufügen. Diese Regeln werden direkt von den Eigenschaften äquivalenter Zustandspaare und unerreichbarer Zustände abgeleitet.

Table of Contents

Chapter 1

Introduction

Automata theory is recommended as part of a standard computer science curriculum [12, pp. 5-6]. As other such theories it provides the chance to gain a precise cognitive model yielding new perspectives on problems and givens. This may thus lead to increased problem solving skills and more accurate thinking.

A typical task in automata theory is the minimization of a given deterministic finite automaton (DFA). The classic textbook "Introduction to automata theory, languages, and computation" by Hopcroft et. al. [15] presents a practicable minimization algorithm. In this work we will confine ourselves to look at DFA minimizations using that algorithm.

In an introduction course to theoretical computer science minimization tasks are thus likely to occur in supplementary exercises or exams. As of the creation of such tasks, one may assume, that it is done mostly manually. Automation would yield here the following advantages:

- freeing time for other things, e.g. research, helping students face-to-face, designing the whole exercise sheet
- generation of tasks which lie in a well-defined range
- increased predictability and consistency of the generated task properties, which can be adjusted accurately through various parameters
- saves human operators from the generating task which involves monotonous work

Gregor: Delete or find externally from Wikipedia Engagement on this topic promises moreover increased clarification which kind of minimization tasks can be generated, and where difficulties of those tasks lie.

This work aims to provide theoretical foundations for a DFA minimization task generator. What requirements a user has towards such a program will be discussed in a short requirements analysis. Based on this work a DFA minimization generator will be devised. Alongside to this thesis an implementation of such a generator has been developed. It can be found at `https://github.com/bt701607/Generation-of-DFA-Minimization-Problems`.

Chapter 2

Problem definition and approach

In this chapter we will set foundations, investigate sensible parameters and requirements for a minimization task generator and deduce our general approach to build such a program.

2.1 Preliminaries

We start with defining preliminary theoretical foundations. By $[[n]]$ we will denote the set of integers $\{0, \dots, n-1\}$.

2.1.1 Deterministic Finite Automatons

A 5-tuple $A = (Q, \Sigma, \delta, s, F)$ with Q being a finite set of *states*, Σ a finite set of *alphabet symbols*, $\delta\colon\ Q \times \Sigma \to Q$ a *transition function*, $s \in Q$ a *start state* and $F \subseteq Q$ *final states* is called *deterministic finite automaton* (DFA) [15, p. 46]. From now on $\mathcal{A}$ shall denote the set of all DFAs.

We say $\delta(q, \sigma) = p$ is a transition from q to p using symbol σ. We define the *extended transition function* $\delta^* : Q \times \Sigma^* \to Q$ of a DFA $A = (Q, \Sigma, \delta, s, F)$ as:

- $\delta^*(q, \varepsilon) = q$
- $\delta^*(q, w\sigma) = \delta(\delta^*(q, w), \sigma)$ for all $q \in Q$, $w \in \Sigma^*$, $\sigma \in \Sigma$

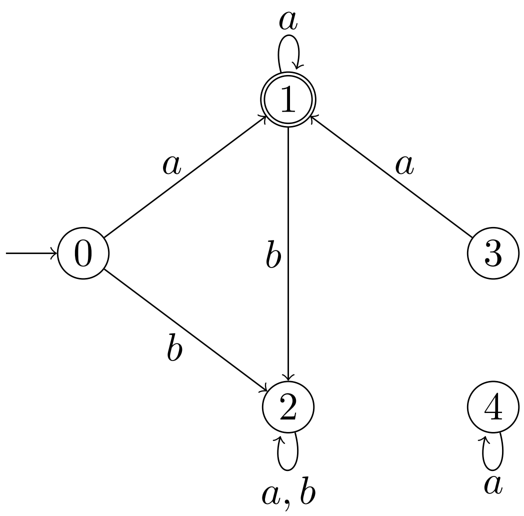
Then, the *language* of A is defined as $L(A) = \{\ w \mid \delta^*(w) \in F\ \}$ [15, pp. 49-50. 52].

Given a state $q \in Q$. With $d^-(q)$ we denote the set of all *ingoing* transitions $\delta(q', \sigma) = q$ of q. With $d^+(q)$ we denote the set of all *outgoing* transitions $\delta(q, \sigma) = q'$ of q [9, pp. 2-3].

Definition 1 ((Un-)Reachable State)**.** We say a state q is *(un-)reachable* in a DFA A, iff there is (no) a word $w \in \Sigma^*$ such that $\delta^*(s, w) = q$.

If all states of a DFA A are reachable, then we call A *accessible* [9, p. 2].

A DFA is called *complete* iff for all states, every symbol of the alphabet is used on an outgoing transition: $\forall q \in Q\colon \forall \sigma \in \Sigma\colon \exists p \in Q\colon \delta(q, \sigma) = p$. Note, that every incomplete DFA can be converted to a complete one by adding a so called *dead state* [15, p. 67]. The resulting automaton has the same language.



$A = (Q, \Sigma, \delta, s, F)$

$Q = [[5]] = \{\ 0, 1, 2, 3, 4\ \}$
$\Sigma = \{\ a, b\ \}$
$\delta = \{\ ((0, a), 1), ((0, b), 1), \ldots\ \}$
$s = 0$
$F = \{\ 1\ \}$

Figure 2.1: An example DFA. The states 3 and 4 are unreachable. This DFA is not complete since the transitions $\delta(2, b)$ and $\delta(3, b)$ are not defined.

Definition 2 (Minimal DFA)**.** We call a DFA A *minimal*, if there exists no other automaton with the same language using less states.

With $\mathcal{A}_{min}$ we shall denote the set of all minimal DFAs.

Definition 3 (Equivalent and Distinguishable State Pairs)**.** [15, p. 154] A state pair $q_1, q_2 \in Q$ of a DFA $A = (Q, \Sigma, \delta, s, F)$ is called *equivalent*, iff $\sim_A (q_1, q_2)$ is true, whereas

$$q_1 \ \sim_A \ q_2 \Leftrightarrow_{def} \ \forall z \in \Sigma^*\colon \ (\delta^*(q_1, z) \in F \Leftrightarrow \delta^*(q_2, z) \in F)$$

If $q_0 \not\sim_A q_1$, then q_0 and q_1 are called a *distinguishable* state pair. The relation $\sim_A$ is an equivalence relation

2.1.2 Isomorphy of DFAs

Given two DFAs $A_1 = (Q_1, \Sigma_1, \delta_1, s_1, F_1)$ and $A_2 = (Q_2, \Sigma_2, \delta_2, s_2, F_2)$. We say A_1 and A_2 are *isomorph*, iff:

- $|Q_1| = |Q_2|$, $\Sigma_1 = \Sigma_2$ and
- there exists a bijection $\pi\colon Q_1 \to Q_2$ such that:
  $\pi(s_1) = s_2$
  $\forall q \in Q_1\colon (q \in F_1 \iff \pi(q) \in F_2)$
  $\forall q \in Q_1\colon \forall \sigma \in \Sigma_1\colon \pi(\delta_1(q, \sigma)) = \delta_2(\pi(q), \sigma))$

In [24, p. 45] we can find the following statement:

Theorem 1. *Every minimal DFA is unique (has a unique language) except for isomorphy.*

We describe a simple isomorphism test for DFAs in the appendix A.

2.1.3 The minimization algorithm

This minimization algorithm MinimizeDFA works in four major steps, removing essentially states in such a way, that no unreachable states and no equivalent state pairs are left.

1. Compute all unreachable states via breadth-first search.

   function ComUnreachables(A)
   $U \leftarrow Q \setminus \{s\}$ ▷ undiscovered states
   $O \leftarrow \{s\}$ ▷ observed states
   $D \leftarrow \{\}$ ▷ discovered states
   while $|O| > 0$ **do**
   $N \leftarrow \{\ p \mid \exists q \in O\ \sigma \in \Sigma\colon\ \delta(q,\sigma) = p\ \wedge\ p \notin O \cup D\ \}$
   $U \leftarrow U \setminus N$
   $D \leftarrow D \cup O$
   $O \leftarrow N$
   return U

2. Remove all unreachable states and their transitions.

   function RemUnreachables(A, U)
   $\delta' \leftarrow \delta \setminus \{\ ((q,\sigma),p) \in \delta \mid q \in U\ \vee\ p \in U\ \}$
   return $(Q \setminus U, \Sigma, \delta', s, F \setminus U)$

3. Compute all equivalent state pairs ($\sim_A$). Inspired by Schöning [24, p. 46] and Martens [26, p. 17].

   function ComEquivPairs(A)
   $M \leftarrow \{(p,q),(q,p) \mid p \in F, q \notin F\}$
   do
   $M' \leftarrow \{(p,q) \mid (p,q) \notin M \wedge \exists \sigma \in \Sigma\colon (\delta(p,\sigma),\delta(q,\sigma)) \in M\}$
   $M \leftarrow M \cup M'$
   while $M' \neq \emptyset$
   return $Q^2 \setminus M$

   Note that ComEquivPairs requires its input automaton to be complete. **Gregor:** Why?

4. Merge all equivalent state pairs, which are exactly those in $\sim_A$. Inspired by Högberg [17, p. 10].

   function RemEquivPairs(A, $\sim_A$)
   $Q_E \leftarrow \emptyset$
   $\delta_E \leftarrow \emptyset$
   $F_E \leftarrow \emptyset$
   for q **in** Q **do**
   Add $[q]$ to Q_E ▷ $[\cdot]_{\sim_A}$ shall be abbreviated $[\cdot]$
   for σ **in** Σ **do**
   $\delta_E([q],\sigma) = [\delta(q,\sigma)]$
   if $q \in F$ **then**
   Add $[q]$ to F_E
   return $(Q_E, \Sigma, \delta_E, [s], F_E)$

   Note that RemEquivPairs creates complete automatons.

function MinimizeDFA(A)
$A' \leftarrow$ RemUnreachables(A, ComUnreachables(A))
return RemEquivPairs(A', ComEquivPairs(A'))

This DFA minimization algorithm has been found by Hopcroft [14].

Theorem 2. [15, pp. 162-164] MinimizeDFA *computes a minimal DFA to its input DFA.*

When looking at ComEquivPairs, one notes, that it computes distinct subsets of $Q \times Q$ on the way. Indeed, one could write the algorithm in such a way, that these subsets are explicitly computed in form of a function $m\colon \mathbb{N} \to \mathcal{P}(Q \times Q)$:

function m-ComEquivPairs(A)
    $i \leftarrow 0$
    $m(0) \leftarrow \{(p,q),(q,p) \mid p \in F, q \notin F\}$
    do
        $i \leftarrow i + 1$
        $m(i) \leftarrow \{(p,q),(q,p) \mid (p,q) \notin \bigcup m(\cdot) \wedge \exists \sigma \in \Sigma\colon (\delta(p,\sigma),\delta(q,\sigma)) \in m(i-1)\}$
    while $m(i) \neq \emptyset$
    return $\bigcup m(\cdot)$

Using this redefinition, we can easier refer to the state pairs marked in a certain iteration. We will use both variants in exchange.

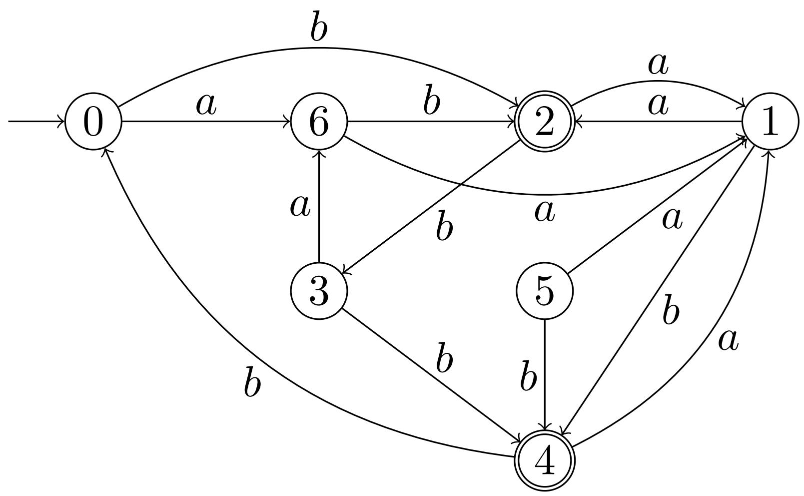
Definition 4. We denote the number of iterations done by ComEquivPairs on an DFA A as $\mathfrak{D}(A)$.

2.2 Requirements analysis

Now that we have introduced all necessary basic definitions, we shall do a short analysis of an example DFA minimization task and its sample solution, as it could have been given to students in an introductory course to automata theory.

Gregor: search for typical task in standard text books

Task: *Consider the below shown deterministic finite automaton A:*



Apply the minimization algorithm and show for each state pair during which ComEquivPairs*-iteration it was marked. Draw the resulting automaton.*

Figure 2.2: An example DFA minimization task.

Figures 2.2 and 2.3 show such a task and solution. The students are confronted with a *task DFA* A_{task}. Firstly, unreachable states have to be eliminated, we then gain A_{re}. Secondly equivalent state pairs of A_{re} are merged such that the minimal *solution DFA* A_{sol} is found.

Solution:
Step 1: Detect and eliminate unreachable states.

State 6 is unreachable.

Step 2: Apply ComEquivPairs *to* A *and merge equivalent state pairs:*

	0	1	2	3	4	6
0	■	1	0		0	2
1	■	■	0	1	0	1
2	■	■	■	0		0
3	■	■	■	■	0	2
4	■	■	■	■	■	0
6	■	■	■	■	■	■

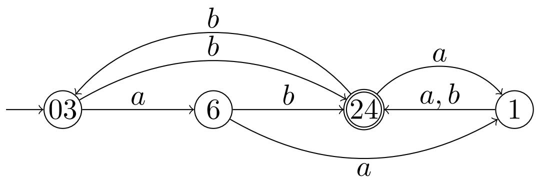


Figure 2.3: Solution to the DFA minimization task in fig. 2.2.

The table T displayed in figure 2.3 is nothing else but a visualization of the function m of m-ComEquivPairs, whereas $T(q_0, q_1) = i \Leftrightarrow (q_0, q_1) \in m(i)$.

We do some rather formal statements and requirements. Firstly, we can state that

- $A_{re} = \text{RemUnreachables}(A_{task}, \text{ComUnreachables}(A_{task}))$ and
- $A_{sol} = \text{RemEquivPairs}(A_{re}, \text{ComEquivPairs}(A_{re}))$

Therefore A_{sol} is minimal regarding A_{re} and A_{task}. Secondly the languages of A_{task}, A_{re} and A_{sol} are equal. We know that ComEquivPairs requires A_{re} to be complete and that RemEquivPairs creates complete DFAs, so A_{sol} is complete too. Furthermore we know that every state of A_{re} is reachable since it is the output of RemUnreachables.

2.2.1 Difficulty adjustment possibilities and sensible requirements

Concerning the execution of MinimizeDFA we find that its difficulty can be classified through various classification numbers. Furthermore we can note some sensible requirements.

ComEquivPairs-depth ($\mathfrak{D}(A_{task})$). Consider the computation of the sets $m(i)$ in ComEquivPairs. Determining $m(0)$ is quite straightforward, because it consists simply of tests whether two states are in $F \times Q \setminus F$ (see 0, line 3). Determining $m(1)$ is less easy: The rule for determining all $m(i), i > 0$ is different to that for $m(0)$ and more complicated (see 0, line 6). Determining $m(2)$ requires the same rule. It shows nonetheless a students understanding of the terminating behavior of ComEquivPairs: It does not stop after computing $m(1)$, but only when no more distinguishable state pairs were found. Concerning the sets $m(i), i > 2$ however no additional understanding can be shown.

It would therefore be sensible if $\mathfrak{D}(A_{task})$ could be adjusted for example by parameters m_{min}, m_{max} which give lower and upper bounds for that value.

Number of states (n_s, n_e, n_u). To control the number of states in A_{task}, A_{re} and A_{sol}, we will introduce three parameters: $n_s, n_e, n_u \in \mathbb{N}$. These parameters get their meaning by the following equations:

$$
\begin{aligned}
|Q_{sol}| &= n_s \\
|Q_{re}| &= n_s + n_e \\
|Q_{task}| &= n_s + n_e + n_u
\end{aligned}
$$

It is sensible to have $n_u > 1, n_e > 1$, such that REMUNREACHABLES and REMEQUIVPAIRS will not be skipped. To not make the task trivial, $n_s > 2$ is sensible. An exercise instructor will find it useful, to control exactly how big n_u, n_e and n_s are: The higher n_u, n_e, the more states have to be eliminated and merged. The higher $n_s + n_e$, the more state pairs have to be checked during COMEQUIVPAIRS.

Alphabet size (k). The more symbols the alphabet of A_{task}, A_{re} and A_{sol} has (note how MINIMIZEDFA does not change the alphabet), the more transitions have to be followed when checking whether $(\delta(q,\sigma), \delta(p,\sigma)) \in m(i-1)$ is true for each state pair p, q.

Number of final states (n_F). Since most DFAs in teaching have about 1 to 3 final states, so being able to set a number of final states, that is familiar to students, might be a reason.

Uniqueness of A_{sol}. For example for an exam it would be sensible to be able to generate a task, so A_{sol} and A_{task}, that has not been generated before. We will use the criterion of isomorphy to distinguish a possible solution DFA from all previously generated ones.

This is sensible since isomorphy distinguishes two minimal DFAs, if and only if they have different languages (see theorem 1).

Note that, if A_{sol} is indeed *new* in that sense, then A_{task} will automatically have a unique language too, since A_{sol} and A_{task} have the same language and this language was then never used before in this context.

Completeness of A_{task}. Even though COMUNREACHABLES and REMUNREACHABLES do not require their input DFA A_{task} to be complete, it is sensible to build it that way. The implications of the completeness-property are - in comparison to the other concepts involved here - rather subtle. This is especially due to its purely representational nature, a DFA has the same language and $\mathfrak{D}$-value, whether it is represented in its complete form or not. Nonetheless we shall introduce a parameter c, that determines if there exist unreachable states, that make A_{task} incomplete. Thus an exercise lecturer could showcase this matter on a DFA and generate according exercises.

Planar drawing of A_{task}. A graph G is *planar* if it can be represented by a drawing in the plane such that its edges do not cross. Such a drawing is then called *planar drawing* of G. A visual aid for students would be given, if the task DFA were planar and presented as a planar drawing. In this work libraries and parameters $p_1, p_2 \in \{0, 1\}$ (toggling planarity of A_{sol}, A_{task}) will be used to allow the option of planarity, but neither ensuring planarity nor planar drawing will be investigated further theoretically.

Maximum degree of any state in A_{task}. The *degree* $deg(q)$ of a state $q \in Q$ in a DFA A is defined as $deg(q) = |d^-(q)| + |d^+(q)|$, so the total number of transitions in which q participates. By capping the maximum degree for all states, the graphical representation of the DFA would be more clear. In this work the inclusion of a maximum degree parameter is omitted.

2.3 Approach and general algorithm

In this work we will first build the solution DFA (step 1), and - based on that - the task DFA by creating equivalent states and adding unreachable states (step 2). Both steps will fulfill all criteria chosen above and are covered in depth in chapter 2 respectively chapter 3.

We will see that $\mathfrak{D}$ and L of both DFAs will be set when building A_{sol}. We know that creating equivalent states and adding unreachable does not change $L(A_{task})$ in comparison to A_{sol}, else MINIMIZEDFA would not work (a minimal DFA has in particular the same language as the original DFA). However we must ensure, that adding those states does not change $\mathfrak{D}$. Since unreachable states are eliminated before COMEQUIVPAIRS is applied, we need only to prove, that creating equivalent states does not change the $\mathfrak{D}$-value. We will do this during the discussion of step 2, more specifically in section 4.1.4.

At the beginning of chapter 2 and 3, we will provide formal problem definitions for both steps, that specify precisely all requirements. Here we shall content ourselves with the definition of the main algorithm:

```
function GENERATEDFAMINIMIZATIONTASK(n_s, k, n_F, m_min, m_max, p_1, p_2, n_e, n_u, c)
    A_sol ← GENERATENEWMINIMALDFA(n_s, k, n_F, m_min, m_max, p_1)
    A_task ← EXTENDMINIMALDFA(A_sol, p_2, n_e, n_u, c)
    return A_sol, A_task
```

Chapter 3

Generating minimal DFAs

We seek algorithms for generation of minimal DFAs that fulfilling the conditions defined in the requirements analysis section 2.2.1. We formally subsume these conditions via the GenerateNewMinimalDFA-problem:

Definition 5 (GenerateNewMinimalDFA)**.**
<u>*Given:*</u>

$$\begin{aligned} n_s &\in \mathbb{N} && \textit{number of states} \\ k &\in \mathbb{N} && \textit{alphabet size} \\ n_F &\in \mathbb{N} && \textit{number of final states} \\ m_{min}, m_{max} &\in \mathbb{N} && \textit{lower and upper bound for } \mathfrak{D}\textit{-value} \\ p &\in \{0,1\} && \textit{planarity-bit} \end{aligned}$$

<u>*Task:*</u> *Compute, if it exists, a solution DFA A_{sol} with*

- $|Q_{sol}| = n_s$, $|\Sigma_{sol}| = k$, $|F_{sol}| = n_F$
- $m_{min} \leq \mathfrak{D}(A_{sol}) \leq m_{max}$
- A_{sol} *being planar iff* $p = 1$
- A_{sol} *being new*

We consider different approaches to solve this problem, of which those using trial-and-error will be discussed most broadly.

Remark. Note that for all generated DFAs we are going to set $Q_{sol} = [[n_s]] = \{0, \ldots, n_s - 1\}$, $\Sigma_{sol} = [[k]] = \{0, \ldots, k-1\}$ and $s_{sol} = 0$, so every DFA of same state number and alphabet size will have the same states and symbols.

As a consequence the presented algorithms will not be able to compute all of $\mathcal{A}_{min}$.

3.1 Using a rejection algorithm

We describe a procedure that is essentially a *rejection algorithm* adjusted to find solution DFAs. The approach works as follows:

Firstly a *test* DFA A_{test} is generated by use of either randomness or enumeration. Alphabet size and number of (final) states will already be correct. On this DFA then tests will be executed, to check if it is minimal, planar (if wished) and new. If this is the case, A_{test} will be returned, if not, new test DFAs are generated until all tests pass.

A note on the search space. If we would not restrict ourselves to $Q_{sol} = [[n_s]]$ and $\Sigma_{sol} = [[k]]$, then for a given number of states and symbols, the number of possible state sets and alphabets would be infinite. This way however we do not have to iterate through infinitely many same-sized versions of Q_{sol} respectively Σ_{sol}. Since there is a finite number of possible transitions functions and final state sets given n_s, k, we can now even guarantee that our algorithm terminates.

1: **function** BUILDNEWMINIMALDFA-1 $(n_s, k, n_F, m_{min}, m_{max} \in \mathbb{N}, p \in \{0, 1\})$
2:     **while** True **do**
3:         generate DFA A_{test} with $|Q|, |\Sigma|, |F|$ matching n_s, k, n_F
4:         **if** A_{test} not minimal **or not** $m_{min} \leq \mathfrak{D}(A_{test}) \leq m_{max}$ **then**
5:             **continue**
6:         **if** $p = 1$ **and** A_{test} is not planar **then**
7:             **continue**
8:         **if** A_{test} is not new **then**
9:             **continue**
10:         **return** A_{test}

We will complete this algorithm by resolving how the tests in lines $4, 6$ and 8 work and by showing two methods for generation of automatons with given restrictions of $|Q|, |\Sigma|$ and $|F|$.

3.1.1 Ensuring A_{test} is minimal and $\mathfrak{D}(A_{test})$ is correct

In order to test, whether A_{test} is minimal, we could simply use the minimization algorithm and compare the resulting DFA and A_{test} using an isomorphy test. However it is sufficient to ensure, that no equivalent or unreachable states exist.

To get $\mathfrak{D}(A_{test})$, we have to run COMEQUIVPAIRS entirely anyway. Hence we can combine the test for equivalent states with computing the DFAs $\mathfrak{D}$-value:

1: **function** HASEQUIVALENTSTATES(A)
2:     $depth \leftarrow 0$
3:     $M \leftarrow \{(p, q), (q, p) \mid p \in F, q \notin F\}$
4:     **do**
5:         $depth \leftarrow depth + 1$
6:         $M' \leftarrow \{(p, q) \mid (p, q) \notin M \wedge \exists \sigma \in \Sigma\colon (\delta(p, \sigma), \delta(q, \sigma)) \in M\}$
7:         $M \leftarrow M \cup M'$
8:     **while** $M' \neq \emptyset$
9:     $hasDupl \leftarrow |\{(p, q) \mid p \neq q \wedge (p, q) \notin M\}| > 0$
10:     **return** $hasDupl, depth$

Since COMEQUIVPAIRS basically computes all distinguishable state pairs $\not\sim_A$, we test in line 9, whether there is a pair of distinguishable states not in $\not\sim_A$.

Regarding the unreachable states, we can just use COMUNREACHABLES and test whether the computed set is empty:

1: **function** HASUNREACHABLESTATES(A)
2:     **return** $|$COMUNREACHABLES$(A)| > 0$

3.1.2 Ensuring A_{test} is planar

There exist several algorithms for planarity testing of graphs. In this work, the library *pygraph*[1] has been used, which implements the Hopcroft-Tarjan planarity algorithm. More

[1] `https://github.com/jciskey/pygraph`

information on this can be found for example in this [18] introduction from William Kocay. The original paper describing the algorithm is by Hopcroft and Tarjan [13].

3.1.3 Ensuring A_{test} is new

In our requirements we stated, that we wanted the generated solution DFA to be new, meaning not isomorph to any previously generated solution DFA. This implies the need of a database, that allows saving and loading DFAs. We name this database *DB1*. Assuming the database is relational, the following scheme is proposed:

$$|Q_A| \quad |\Sigma_A| \quad |F_A| \quad \mathfrak{D}(A) \quad isPlanar(A) \quad encode(A)$$

With this scheme we can fetch once all DFAs matching the search parameters. Thus we need not fetch all previously found DFAs every time, but only those that are relevant. Afterwards we must only check whether any isomorphy test on the current test DFA and one of the fetched DFAs is positive. If any test DFA passes all tests and is going to be returned, then we have to save that DFA in the database.

A more concrete specification of this proceeding is shown below, embedded in the main algorithm:

function BuildNewMinimalDFA-2 $(n_s, k, n_F, m_{min}, m_{max}, p)$
    $l \leftarrow$ all DFAs in DB1 matching $n_s, k, n_F, m_{min}, m_{max}, p$
    while True **do**
        ...
        if A_{test} is isomorph to any DFA in l **then**
            continue
        save A_{test} and its respective properties in DB1
        return A_{test}

To test whether A_{test} is isomorph to any found DFA, we use the isomorphism test described in section 2.1.2.

3.1.4 Option 1: Generating A_{test} via Randomness

We now approach the task of generating a random DFA whereas alphabet and number of (final) states are set. For our generated DFA we choose Q_{sol}, Σ_{sol} and the start state as explained in remark 3.

The remaining elements that need to be defined are δ and F. The set of final states is supposed to have a size of n_F and be a subset of Q. Therefore we can simply choose randomly n_F distinct states from Q.

The transition function has to make the DFA complete, so we have to choose an "end" state q' for every state-symbol-pair q, σ in $Q \times \Sigma$. There is no restriction concerning q', so we can randomly choose $\delta(q, \sigma) = q'$ from Q.

With defining how to compute δ we have covered all elements of a DFA.

function BuildNewMinimalDFA-3a $(n_s, k, n_F, m_{min}, m_{max}, p)$
    $l \leftarrow$ all DFAs in DB1 matching $n_s, k, n_F, m_{min}, m_{max}, p$
    $Q \leftarrow [[n_s]]$
    $\Sigma \leftarrow [[k]]$
    while True **do**
        $\delta \leftarrow \emptyset$

for q **in** Q **do**
  for σ **in** Σ **do**
    $\delta(q, \sigma)$ = random chosen state from Q
$s \leftarrow 0$
$F \leftarrow$ random sample of n_F states from Q
$A_{test} \leftarrow (Q, \Sigma, \delta, s, F)$
if A_{test} not minimal **or not** $m_{min} \leq \mathfrak{D}(A_{test}) \leq m_{max}$ **then**
  continue
if $p = 1$ **and** A_{test} is not planar **then**
  continue
if A_{test} is isomorph to any DFA in l **then**
  continue
save A_{test} and its respective properties in DB1
return A_{test}

3.1.5 Option 2: Generating A_{test} via Enumeration

The second method of test DFA generation is based on the idea, that instead of randomly generating F and δ, we could just enumerate through all possible final state sets and transition functions.

Both enumerations are finite, given n_s and k. Having a requirement of n_F final states, then $\binom{n_s}{n_F}$ is the number of possible F-configurations. On the other hand there are ${n_s}^{n_s k}$ possible δ-configurations: We have to choose one of n_s possible end states for every combination in $Q \times \Sigma$ - so $n_s k$ times.

Again we will call our states and symbols $[[n_s]]$ resp. $[[k]]$. We will represent the state of an enumeration with two fields F_F and F_δ. The first field shall have n_s Bits, whereas Bit $F_F[i] \in \{0, 1\}$ represents the information, whether i is a final state or not. The second field shall have $n_s k$ entries containing state names, such that entry $F_\delta[i * k + j] = q, q \in [[n_s]]$ says, that $\delta(i, j) = q$.

Example 1. *Given $n_s = 4$, $k = 2$, $n_F = 3$. Note that for the sake of readability we will use $a, b, \ldots$ instead of $0, 1, 2, \ldots$ as alphabet symbols. An example F_F-array could be 1101. Since $F_F[i]$ is 1 for $i = 0, 1, 3$ the final states are:*

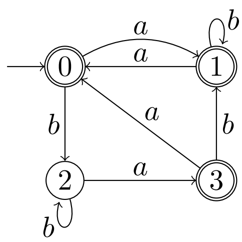
$$F = \{\ 0, 1, 3\ \}$$

A possible F_δ-array could be 12013201. The following table depicts, how F_δ assigns one state to every combination of states and symbols $(q, \sigma) \in Q \times \Sigma$ and thus defines δ:

$q \in Q$	0		1		2		3	
$\sigma \in \Sigma$	a	b	a	b	a	b	a	b
$\delta(q, \sigma)$	1	2	0	1	3	2	0	1

$\delta(0, a) = 1$, $\delta(0, b) = 2$, $\delta(1, a) = 0, \ldots$

The corresponding DFA might then look like this:



□

Given an enumeration state F_F, F_δ and n_s, k, n_F we will then compute the next DFA based on this state as follows. We will treat both fields as numbers, F_F as 2-ary and F_δ as n_s-ary. To get to the next DFA, we will first increment F_δ by 1. If $F_\delta = n_s - 1 \dots n_s - 1$, then we increment F_F until it contains n_F ones (again) and set F_δ to $0 \dots 0$. This behavior is summarized in the following algorithm:

function IncrementEnumProgress $(F_F, F_\delta, n_s, k, n_F)$
    add 1 to $(F_\delta)_{n_s}$
    if $F_\delta = 0 \dots 0$ **then**
        while $\#_1(F_F) \neq n_F$ **do** ▷ if the number of 1s in F_F is not n_F
            add 1 to $(F_F)_2$
            if $F_F = 0 \dots 0$ **then**
                return $\perp$
            $F_\delta = 0 \dots 0$
    return F_F, F_δ

In this algorithm we assume, that adding 1 to a n-ary number $n-1\ n-1\ \dots n-1$ yields $00 \dots 0$.

Example 2. *We showcase a sample enumeration at points, that demonstrate the semantics of different increments. We will use $n_s = 4$, $k = 2$ and $n_F = 2$. Note that we will use $a, b, \dots$ instead of $0, 1, \dots$ as symbols again. Valid enumeration progresses are depicted green.*

(1). We start with the initial enumeration progress. In this case, a simple addition of 1 to F_δ does not cause an overflow of F_δ (2), meaning the enumeration increment is already finished.

(3). In this state however F_δ becomes $0 \dots 0$ (4) after adding 1. Thus we add 1 to F_F, until it contains the required number of ones again (so we always have f final states). The next 4-ary number with 2 ones after 0011 is 0101 (5).

(6). Here F_δ is at its maximum and there is no higher 4-ary number with 2 ones. So if the algorithm is now applied and tries to get the next valid F_F-value, it will eventually reach (7), which indicates the enumeration has found its end.

(1) $(0011)_2$ $(00\ 00\ 00\ 00)_4$
$+1$
(2) $(0011)_2$ $(00\ 00\ 00\ 01)_4$
$\dots$ $+x$
(3) $(0011)_2$ $(33\ 33\ 33\ 33)_4$
(4) $(0100)_2$ $(00\ 00\ 00\ 00)_4$ $+1$
(5) $(0101)_2$ $(00\ 00\ 00\ 00)_4$
$\dots$ $+x$
(6) $(1100)_2$ $(33\ 33\ 33\ 33)_4$
$\dots$
(7) $(1111)_2$ $(33\ 33\ 33\ 33)_4$

□

Based on the incremented bit-fields the new DFA can be build according to the semantics defined above:

function DFAfromEnumProgress $(F_F, F_\delta, n_s, k, n_F)$
    $Q \leftarrow [[n_s]]$
    $\Sigma \leftarrow [[k]]$
    $\delta \leftarrow \emptyset$
    for i **in** $[[n_s]]$ **do**
        for j **in** $[[k]]$ **do**
            $\delta(i, j) = F_\delta[i * k + j]$
    $s \leftarrow 0$
    for i **in** $[[n_s]]$ **do**

if $F_F[i] = 1$ **then**
Add i to F
return $(Q, \Sigma, \delta, s, F)$

The initial field values are each time $0 \dots 0$. Note how construction and use of these fields results in DFAs with correct alphabet size and number of (final) states. An enumeration can finish either because a matching DFA has been found or all DFAs have been enumerated. The latter is the case, if $F_F = 1 \dots 1$ and $F_\delta = n_s - 1 \dots n_s - 1$.

Once the enumeration within a call of BuildNewMinimalDFA has been finished, it is reasonable to *save* the enumeration progress (meaning the current content of F_F, F_δ), such that during the next call enumeration can be resumed from that point on. The alternative would mean, that the enumeration is run in its entirety until that point again, whereas all so far found DFAs would be found to be not new. Thus we introduce a second database $DB2$ with the following table:

$$|Q_A| \quad |\Sigma_A| \quad F_F \quad F_\delta$$

We reduce the enumeration room for each calculation.

function BuildNewMinimalDFA-3b $(n_s, k, n_F, m_{min}, m_{max}, p)$
$l \leftarrow$ all DFAs in DB1 matching $n_s, k, n_F, m_{min}, m_{max}, p$
$F_F, F_\delta \leftarrow$ load enumeration progress for n_s, k, n_F, p from DB2
while True **do**
if F_F, F_δ is finished **then**
save F_F, F_δ
return $\perp$
$A_{test} \leftarrow$ next DFA based on F_F, F_δ
if A_{test} not minimal **or not** $m_{min} \leq \mathfrak{D}(A_{test}) \leq m_{max}$ **then**
continue
if $p = 1$ **and** A_{test} is not planar **then**
continue
if A_{test} is isomorph to any DFA in l **then**
continue
save F_F, F_δ in DB2
save A_{test} and its respective properties in DB1
return A_{test}

3.2 Alternative approach: Building $m(i)$ bottom up

Build m from m-ComEquivPairs iteratively. (Why would this basically result in running ComEquivPairs all the time?)

3.3 Related research on DFA generation

Nicaud provides an overview of results on random generation and combinatorial properties of DFAs in [21]. We will outline relevant related research.

Nicaud's summary indicates, that research has focused on randomized generation of accessible, but not minimal DFAs so far. In the following we will sketch some approaches that have come up.

Using the recursive method. Champarnaud and Paranthoën [9] continue ideas started by Nicaud in his thesis [20]. Let $\mathfrak{F}_{\mathfrak{n},\mathfrak{m}}$ be the set of extended m-ary trees of order n. These trees are characterized by a partitioning $V = N \uplus L$ with $|N| = n$ and $v \in N \Rightarrow d^+(v) = m$ and $v \in L \Rightarrow d^+(v) = 0$. We define the following set of tuples using $s = n(m-1)$:

$$\mathfrak{R}_{\mathfrak{m},\mathfrak{n}} = \{\ (k_1, \ldots, k_s) \in \mathbb{N}^s \mid \forall i \in [2, s] \colon k_i \geq \left\lceil \frac{i}{m-1} \right\rceil \ \textit{and}\ k_i \geq k_{i-1}\ \}$$

In [9, p. 6] it is shown that there exists a bijection φ between $\mathfrak{F}_{\mathfrak{n},\mathfrak{m}}$ and $\mathfrak{R}_{\mathfrak{m},\mathfrak{n}}$ which maps to k_i, $i \in [1, s]$ of a tuple the number of leaves visited before the ith leaf in a tree. The connection to accessible DFAs is established by proving that "transition structures[2]" with $|Q| = n$, $|\Sigma| = m$ reduced to the set of the smallest paths from the s to each other state are in bijection with extended m-ary trees of order n (see [9, p. 8]).

As a consequence they are able to construct a random generation of accessible complete DFAs using the "recursive method" from [22] which generates n-tuples [9, p. 10]. Nicaud states in his survey that the algorithm's runtime is $\mathcal{O}(n^2)$ but notes, that generation of DFAs with more than "a few thousand states" is practically hard to do [21, pp. 10-11].

Almeida et. al. [1, 2, 23] present and implement methods using a string-encoding of DFAs for exact enumeration and random generation of DFAs. Nicaud [21, p. 11] states in a remark, that this approach uses the same recursive method and differs only in the DFA encoding.

Using Boltzmann sampler. Bassino, David and Nicaud present and implement a more efficient random generator of accessible complete DFAs in [4, 6]. Their idea is based on so called Boltzmann samplers. This framework of samplers is characterized in particular by the fact that the size of its generated objects are not fixed but in an interval around a given input size - this stands in opposition to most random generators in literature [11, p. 2].

In [6] the authors use a Boltzmann sampler to generate set partitions that are shown to be in bijection with so called box diagrams [6, p. 8] which are in turn in bijection to accessible complete DFAs [6, p. 4]. They thus acquire an average runtime complexity of $\mathcal{O}(n^{3/2})$ for a single random generation.

Using a rejection algorithm. Carayol and Nicaud [8] give a simple algorithm with the same runtime complexity. They use a result stating that the size of accessible DFAs is concentrated around some computable value. In the end random possibly inaccessible DFAs of a specific size are generated, of which afterwards all unreachable states are deleted. This is thus essentially a rejection algorithm with clever generation of test DFAs. They furthermore show that allowing approximate sampling with the number of states being in $[n - \varepsilon\sqrt{n}, n + \varepsilon\sqrt{n}]$ results in linear expected runtime.

Others and comparison to algorithm presented in this work. In his survey Nicaud mentions a paper by Bassino and Sportiello [3] that yields random generation of accessible DFAs in expected linear time. This work will not be discussed further here.

In this work we use a rejection algorithm that generates test DFAs either by randomization or by enumeration. Both methods implement a naive approach. The generated test DFAs are not necessary minimal and in particular not necessary accessible as in [8]. The enumeration method uses encodings of DFAs similar to those used by Almeida et. al. [23].

[2] Those are essentially DFAs without final state sets.

3.4 Empirical and combinatorial results

Concerning combinatorial properties of DFAs, several authors (e.g. [6, 10, 16]) consider a work from Vyssotsky [25] in the Bell laboratories to be the first on this subject. A contribution by Korshunov [19] is often cited in this regard, for he firstly "determines an asymptotic estimate of the number of accessible complete and deterministic n-state automata over a finite alphabet" [5].

Implementations (e.g. [1, 4]) of various random and enumeration generation methods have given rise to several empirical observations concerning the number of minimal DFAs, their fraction among all DFAs and so forth.

Domaratzki, Kisman, and Shallit [10] give some asymptotic estimates and explicit computations for the number of each several types of languages and automata that are distinct. The here relevant results have been subsumed and extended in [2, p. 8] by means of exact enumeration and are confirmed in [4].

$\|\Sigma\|$ (k)	$\|Q\|$ (n)	$\|\mathcal{A}_{min,n,k}\|$	$\|\mathcal{A}_{n,k}\|$	Minimal %
$k = 2$	2	**24**	64	0.38
	3	**1028**	5832	0.18
	4	**56014**	1048576	0.05
	5	**3705306**	312500000	0.01
	6	**286717796**	139314069504	0.0
	7	**25493886852**	86812553324672	0.0
$k = 3$	2	**112**	256	0.44
	3	**41928**	157464	0.27
	4	**26617614**	268435456	0.1
	5	**25184560134**	976562500000	0.03
$k = 4$	2	**480**	1024	0.47
	3	**1352732**	4251528	0.32
	4	**7756763336**	68719476736	0.11
$k = 5$	2	**1984**	4096	0.48
	3	**36818904**	114791256	0.32

Figure 3.1: Table depicting the amount of minimal complete DFAs among all complete DFAs for various sizes of Q, Σ. The numbers of minimal DFAs (bold numbers) are taken from [2, p. 8].

In table 3.1 we use these results to determine the ratios of minimal complete DFAs among all complete DFAs for given $|Q|$ and $|\Sigma|$. The number of all DFAs is computed as follows:

$$|\mathcal{A}_{n,k}| = \underbrace{n^{n*k}}_{\#\text{possible } \delta\text{'s}} * \underbrace{2^n}_{\#\text{possible sets } F}$$

Thus we gain an insight into how probable the generation of a distinct minimal test DFA is without applying further constraints. For our proposed default parameters $n \in [4-5]$ and $k \in [2-3]$ the probabilities of successful generation range from 1% to 5%. Practical tests have shown that this leads to sufficient short run times for our implementation.

Further interesting results in this area include the determination of the fraction of minimal automata among accessible complete DFAs [5] and asymptotic estimates for the number of states that a random minimized DFA has [7].

Chapter 4

Extending minimal DFAs

We firstly define a formal problem for extending a minimal DFA A_{sol} to a task DFA A_{task} based on our requirements analysis (see 2.2.1):

Definition 6 (ExtendMinimalDFA)**.**

Given:

$$
\begin{aligned}
A_{sol} = (Q, \Sigma, \delta, s, F) \in \mathcal{A}_{min} &\quad \textit{solution DFA} \\
n_e \in \mathbb{N} &\quad \textit{number of states creating equivalent state pairs} \\
n_u \in \mathbb{N} &\quad \textit{number of unreachable states} \\
p \in \{0, 1\} &\quad \textit{planarity-bit} \\
c \in \{0, 1\} &\quad \textit{completeness-bit}
\end{aligned}
$$

Task: *Compute, if it exists, a task DFA A_{task} with*

- $Q_{task} = Q_{sol} \cup \{r_1, \ldots, r_{n_e}, u_1, \ldots, u_{n_u}\}$
- $r_1, \ldots, r_{n_e}$ *each creating an equivalent state pair*
- $u_1, \ldots, u_{n_u}$ *unreachable*
- $\Sigma_{task} = \Sigma_{sol}$, $s_{task} = s_{sol}$, $F_{task} \subseteq F_{sol}$
- A_{task} *being planar iff* $p = 1$
- A_{task} *being complete iff* $c = 1$
- A_{sol} *being isomorph to* MinimizeDFA(A_{task})

In order to fulfill these requirements we will deduce for both kinds of states how they may be added by examining their desired properties. We will show for the action of adding equivalent states, that this does not change a DFAs $\mathfrak{D}$-value.

4.1 Creating equivalent state pairs

Step 3 and 4 of the minimization algorithm are concerned with detection and elimination of equivalent state pairs. We now want to add states $r_1, \ldots, r_{n_e}$ to a DFA A_{sol}, gaining A_{re} with $Q_{re} = Q_{sol} \cup \{r_1, \ldots, r_{n_e}\}$, such that each of these states is equivalent to a state in A_{re}. Note that, for reasons of clarity, we are going to abbreviate from now on $A_{re} = A$, $Q_{re} = Q$, $\sim_{A_{re}} = \sim_A$ etc.

Consider the properties $r_1, \dots, r_{n_e}$ must have. They are equivalent to states $o_1, \dots, o_{n_e}$ of A.

$$\exists r_1, \dots, r_{n_e} \in Q\colon\ \exists o_1, \dots, o_{n_e} \in Q\colon\ \forall i \in [1, n_e]\colon\ r_i \sim_A o_i$$

But we know also and in particular, that each of them is equivalent a state e of A_{sol}.

$$\exists r_1, \dots, r_{n_e} \in Q\colon\ \forall i \in [1, n_e]\colon\ \exists e \in Q_{sol}\colon\ r_i \sim_A e$$

In our algorithm, we will choose the state e for each state we add.

4.1.1 Adding outgoing transitions

Regarding the outgoing transitions of any r_i equivalent to a state e, we are directly restricted by the relationship $\forall \sigma \in \Sigma\colon [\delta(r_i, \sigma)]_{\sim_A} = [\delta(e, \sigma)]_{\sim_A}$. Thus, when adding some r_i, we have to choose for each symbol $\sigma \in \Sigma$ at exactly one transition (completeness requirement for A) from the following set:

$$O_{e,\sigma} = \{\ ((r_i, \sigma), q) \mid q \in [\delta(e, \sigma)]_{\sim_A}\ \}$$

Since the solution DFA is complete and since every here added state gets a transition for every alphabet symbol, we know that every $O_{e,\sigma} \neq \emptyset$.

Gregor: Why does this not affect the eq. class of any other state?

4.1.2 Adding ingoing transitions

First of all, we know, that r_i is reachable, since every state of A must be reachable, so we need to give r_i at least one ingoing transition. Doing this, we have to ensure, that any state q, that gets such an outgoing transition to r_i remains in its $\sim$-equivalence class.

Thus a fitting state q has to have a transition to some state in $[r_i]_{\sim_A} = [e]_{\sim_A}$ already. So, given a state q with $\delta(q, \sigma) = p$ and $p \in [e]_{\sim_A}$, we can set $\delta(q, \sigma) = r_i$ and thus "steal" q its ingoing transition.

We see here, that q must have at least 2 ingoing transitions, else it would become unreachable. Thus we summarize:

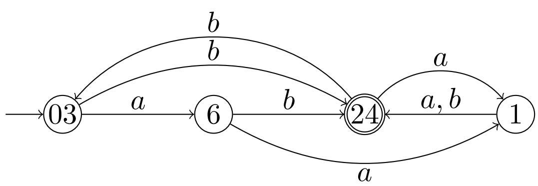
$$I_e = \{\ ((q, \sigma), p) \mid \delta(q, \sigma) = p \land p \in [e] \land d^-(p) \geq 2\ \}$$

Choose at least one $((q, \sigma), p) \in I_e$, remove $((q, \sigma), p)$ from δ and add $((q, \sigma), r_i)$.

These finding lead us to a general requirement regarding the choice of a state e for an r_i: The equivalence class of any e has to contain at least one state with at least 2 ingoing transitions (see fig. 4.1). We establish the following notion to pin down this restriction:

$$duplicatable(q) \Leftrightarrow_{def} (\exists p \in [q]_{\sim_A}\colon |d^-(p)| \geq 2)$$

The number of duplicatable states in any accessible DFA A is 0 for $|\Sigma| \leq 1$ (due to the restriction $|d^-(p)| \geq 2$) and greater than 0 for $|\Sigma| > 1$ due to the pigeonhole principle: An accessible complete DFA has $|Q||\Sigma|$ transitions which have to be spread across $|Q|$ states.



Example 3.

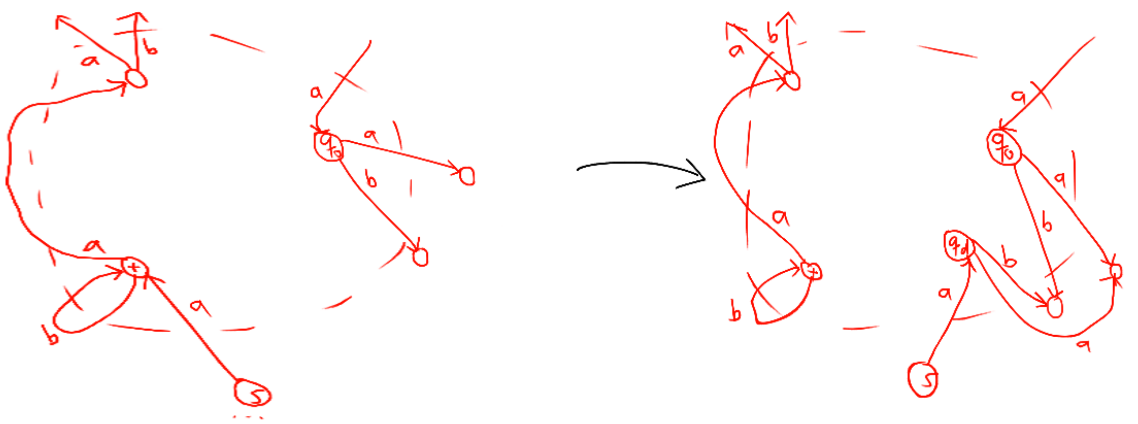


Figure 4.1: If an equivalence class (here denoted by the states in the dashed area) contains a state with 2 or more ingoing transitions (in this case p), then a state equivalent to any of the classes states may be added. Here r is equivalent to o and is "stealing" the ingoing transition $\delta(q, a)$ from p.

4.1.3 The algorithm

function CreateEquivalentStatePairs(A, n_e)
  $Q \leftarrow Q_{sol}$
  $\delta \leftarrow \delta_{sol}$
  $F \leftarrow F_{sol}$
  $K \leftarrow \{ \{q\} \mid q \in Q \}$ ▷ tracks the equivalence classes of A
  $k(q) = C$ such that $q \in C$ and $C \in K$ ▷ returns the equivalence class to q
  $in(q) = |d^-(q)|$ for all $q \in Q$ ▷ tracks the number of ingoing t.
  for i **in** $[1, n_e]$ **do**
    for q **in** Q **do** ▷ find a duplicatable state e
      if $in(q) \geq 2$ **then**
        $e \leftarrow$ random chosen state from $k(q)$
        break
    $r_i \leftarrow$ unused state label ▷ create to e equivalent state r_i
    Add r_i to Q
    Add r_i to $k(e)$
    for σ **in** Σ **do** ▷ add $d^+(r_i)$
      $\delta(r_i, \sigma) =$ random chosen state from $k(\delta(e, \sigma))$
    $P \leftarrow \{ ((s, \sigma), t) \in \delta \mid t \in k(e),\ in(t) \geq 2 \}$ ▷ add $d^-(r_i)$
    $C \leftarrow$ random nonempty subset of P
    for $((s, \sigma), t)$ **in** C **do**
      $in(t) \leftarrow in(t) - 1$
      $in(r_i) \leftarrow 1$
      $\delta(s, \sigma) = r_i$
  return $(Q, \Sigma_{sol}, \delta, s_{sol}, F)$

Note that computing an unused state label can be easily done by e.g. taking the maximum of all solution DFA states (which are nothing else but numbers) and adding one.

4.1.4 Creating equivalent state pairs does not change $\mathfrak{D}$

To prove this statement, we will prove two minor propositions first. In this context we will call a word w *distinguishing word of* p, q, iff $d_A(w, p, q)$ whereas

$$d_A(w, p, q) \Leftrightarrow (\delta^*(p, w) \in F \Leftrightarrow \delta^*(q, w) \notin F)$$

The following lemma and its proof are in parts inspired by [26, ch. 4 p. 18].

Lemma 1. *In the context of* ComEquivPairs *the following is true: If and only if* $(p, q) \in m(n)$*, the shortest distinguishing word of* p, q *has length* n*. Formally:*

$$\begin{aligned}(p, q) \in m(n) \Longleftrightarrow\ & \exists w \in \Sigma^* \colon (|w| = n\ \wedge d_A(w, p, q)) \\ & \wedge \nexists v \in \Sigma^* \colon (|v| < |w|\ \wedge d_A(v, p, q))\end{aligned}$$

Proof. Per induction on the number of ComEquivPairs-iterations n.

$n = 0$, "$\Leftrightarrow$".

$$\begin{aligned}& (p, q) \in m(0) = \{(p, q), (q, p) \mid p \in F, q \notin F\} \text{ (see 0, line 2))} \\ \Leftrightarrow\ & \text{one of } p, q \text{ in } F, \text{ one not} \\ \Leftrightarrow\ & \text{one of } \delta^*(p, \varepsilon), \delta^*(q, \varepsilon) \text{ in } F, \text{ one not} \\ \Leftrightarrow\ & \exists w \in \Sigma^* \colon (|w| = 0 \wedge \text{one of } \delta^*(p, w), \delta^*(q, w) \text{ in } F, \text{ one not}) \\ \Leftrightarrow\ & \exists w \in \Sigma^* \colon (|w| = 0\ \wedge d_A(w, p, q)) \\ & \text{and there cannot be no shorter such word } \checkmark\end{aligned}$$

$0 \ldots n-1 \rightarrow n$, "$\Rightarrow$". Then the following holds for some states p, q (see 0, line 5):

$$(p, q) \in m(n) = \{(p, q), (q, p) \mid (p, q) \notin \bigcup m(\cdot) \wedge \exists \sigma \in \Sigma \colon (\delta(p, \sigma), \delta(q, \sigma)) \in m(n-1)\} \tag{4.1}$$

So in particular there exists a symbol σ such that $(\delta(p, \sigma), \delta(q, \sigma)) \in m(n-1)$. Let $(p', q') = (\delta(p, \sigma), \delta(q, \sigma))$, so $(p', q') \in m(n-1)$.

Per induction there exists a shortest distinguishing word w', $|w'| = n-1$ to p', q'. Thus one of $\delta^*(p', w'), \delta^*(q', w')$ is in F, one not and there is no shorter word.

Thus one of $\delta^*(p, \sigma w'), \delta^*(q, \sigma w')$ is in F, one not, which makes $\sigma w'$ a distinguishing word of length n for p, q.

Since (p, q) is not in any $m(i), i < n$ (recall $(p, q) \notin \bigcup m(\cdot)$ of eq. 4.1), there is per induction no shorter distinguishing word. ✓

$0 \ldots n-1 \rightarrow n$, "$\Leftarrow$". Then the following holds for some states p, q:

$$\begin{aligned}& \exists w \in \Sigma^* \colon (|w| = n\ \wedge d_A(w, p, q)) \\ \wedge\ & \nexists v \in \Sigma^* \colon (|v| < |w|\ \wedge d_A(v, p, q))\end{aligned}$$

So there exists a word w with $|w| = n > 0$ such that one of $\delta^*(p, w), \delta^*(q, w)$ is in F, one not and there is no shorter word fulfilling this property.

Since w is non-empty there exists a symbol σ such that $w = \sigma w'$. Let $(\delta(p, \sigma), \delta(q, \sigma)) = (p', q')$.

Thus, if one of $\delta^*(p, \sigma w'), \delta^*(q, \sigma w')$ is in F and one not, then the same must hold for $\delta^*(p', w'), \delta^*(q', w')$, so w' is a distinguishing word for p', q'.

It is also the shortest one, because, if there existed a shorter word v', $|v'| < |w'|$, then $\sigma v'$ would be a distinguishing word shorter than w for p, q which is contradictory.

Since w' is a shortest distinguishing word for p', q', we may deduce now per induction, that $(p', q') \in m(n-1)$.

The pair (p, q) is not in any $m(i)$, $i < n$, since else-wise per induction the shortest distinguishing word would be shorter than w and thus not w. Since $(p', q') \in m(n-1)$ and $(\delta(p, \sigma), \delta(q, \sigma)) = (p', q')$, we can then deduce by the definition of m, that $(p, q) \in m(n)$. ✓ □

Lemma 2. *If* ComEquivPairs *has done* $\mathfrak{D}(A)$ *iterations and terminated, then the longest word* w*, that is a shortest distinguishing word for any state pair, has length* $\mathfrak{D}(A) - 1$.

Proof. Via direct proof. Assume m-ComEquivPairs(A) has done n iterations (so $\mathfrak{D}(A) = n$). We observe, that

1. $\forall i \in [0, n-1] \colon m(i) \neq \emptyset$
2. $m(n) = \emptyset$
3. $\forall i > n \colon m(i) = \bot$

m-ComEquivPairs(A) terminates iff $m(i) = \emptyset$. If the first point would not hold, then the algorithm would have stopped before. Since the algorithm did n iterations, the internal variable i must be n at the end of the last iteration. And m is not changed after the last iteration; thus follow the second and third point.

We will prove now there exists a shortest distinguishing word of length $n-1$, but the existence of a longer distinguishing word that is shortest for some state pair leads to a contradiction. Recall lemma 1:

$$\begin{aligned}(p, q) \in m(n) \iff &\exists w \in \Sigma^* \colon (|w| = n \ \wedge d_A(w, p, q)) \\ &\wedge \nexists v \in \Sigma^* \colon (|v| < |w| \ \wedge d_A(v, p, q))\end{aligned}$$

Following this lemma and point 1, we can deduce the existence of a shortest distinguishing word w with $|w| = n - 1 = \mathfrak{D}(A) - 1$ for some $p, q \in Q$.

There cannot be any shortest distinguishing word w' with $|w'| = k > n - 1$ for any two states $p', q' \in Q$ fulfilling this property. Following the lemma again, $m(k)$ for some $k > n - 1$ would be defined and non-empty, which is contradictory to point 2 and 3. □

Lemma 3. *If* w *is shortest distinguishing word for* p, q *and* $q \sim_A q'$*, then* w *is a shortest distinguishing word for* p, q'.

Proof. Via direct proof. The relation $q \sim_A q'$ says that for all words z, $\delta^*(q, z)$ and $\delta^*(q', z)$ are both in F or both not in F. Consequently asking whether the following is true

$$\delta^*(p, z) \in F \Leftrightarrow \delta^*(q, z) \in F$$

is the same as asking whether this is true:

$$\delta^*(p, z) \in F \Leftrightarrow \delta^*(q', z) \in F$$

This implies, that q and q' have exactly the same distinguishing words with other states (see definition of distinguishing words). As a consequence they too have the same shortest distinguishing words with other states. □

Theorem 3. *Creating equivalent state pairs in a minimal DFA* A *such that* A' *is gained, does not increase the number of* ComEquivPairs*-iterations:* $\mathfrak{D}(A) \geq \mathfrak{D}(A')$.

Proof. Per contradiction. Let us assume there were n states $r_1, \ldots, r_n$ added to a given minimal DFA $A = (Q, \Sigma, \delta, s, F)$ resulting in a DFA $A' = (Q', \Sigma, \delta', s, F')$ such that:

- $Q' = Q \cup \{r_1, \ldots, r_n\}$
- $\forall i \in [1, n]\colon\ \exists q \in Q\colon\ r_i \sim_{A'} q$
- Towards a contradiction: $\mathfrak{D}(A) < \mathfrak{D}(A')$.

According to lemma 2 the longest shortest distinguishing word w of A has length $|w| = \mathfrak{D}(A) - 1$, while its counterpart w' in A' has length $\mathfrak{D}(A') - 1$. Consequently $|w| < |w'|$.

There exist states $p', q' \in Q'$ such that w' distinguishes p', q' in A'. We differentiate between three cases regarding the belonging of p', q'.

Both p', q' in Q: Since the longest shortest distinguishing word any state pair in Q can have is guaranteed shorter than w', we may conclude that p', q' being in Q and having w' as shortest distinguishing word is a contradiction. ↯

One of p', q' in Q, one in $Q' \setminus Q$: W.l.o.g. $p' \in Q$, $q' \in Q' \setminus Q$. Consequently $q' \in \{r_1, \ldots, r_n\}$. This implies that $q' \sim_{A'} q$ for an $q \in Q$.

By lemma 3 w' is a shortest distinguishing word for p', q too. But now two states p', q of Q again would have a distinguishing word longer than $|w|$. ↯

None of p', q' in Q: Since p', q' both have to be in $Q' \setminus Q = \{r_1, \ldots, r_n\}$, we can find states $p, q \in Q$ equivalent to p', q'. For these states w' would be a shortest distinguishing word, which is contradictory again. ↯ □

4.2 Adding unreachable states

From step 1 of the minimization algorithm we can deduce how to add unreachable states. These can easily be added to a DFA by adding non-start states with no ingoing transitions (see def. 1). Number and nature of outgoing transitions may be arbitrary.

function AddUnreachableStates (A, n_u, c)
    for n_u **times do**
        $q \leftarrow$ unused state label
        $Q \leftarrow Q \cup \{q\}$
        $outSymbols \leftarrow c = 1\ ?\ \Sigma\ :$ random subset of Σ
        $R \leftarrow$ random chosen sample of $|outSymbols|$ states from $Q \setminus \{q\}$
        for σ **in** $outSymbols$ **do**
            $q' \in R$
            $R \leftarrow R \setminus \{q'\}$
            $\delta \leftarrow \delta \cup \{((q, \sigma), q')\}$
    return A

If completeness is demanded ($c = 1$), then we set Σ as set of all symbols, for which a state shall gain outgoing transitions. Else we choose a random subset for each state, such that some unreachable states may miss some outgoing transitions.

Chapter 5

Conclusion

Our intention was to investigate approaches, how DFA minimization task could be generated automatically. Therefore we discussed requirements to such a program and used some of them to formalize the underlying problems. Our approach to solve those problems was to first generate the minimal solution DFA and afterwards the task DFA by adding equivalent states and unreachable DFAs. This structure was derived from Hopcrofts minimization algorithm.

We did the generation of minimal DFAs via a rejection algorithm using either randomized and enumerating; we rejected in particular DFAs with a language which was found already in a previous run. A short overview over research on this topic confirmed our direction but gave outlook to more efficient variants.

On making minimal DFAs non-minimal no results in research was found. The properties of equivalent state pairs and unreachable states however gave precise and easy applicable rules to add such elements.

When building the task DFA, a question arised concerning the number of iterations by the ComEquivPairs-algorithm ($\mathfrak{D}$), which we wanted to be adjustable via parameter. By proving that the $\mathfrak{D}$-value does not change if we extend minimal DFAs, we could ensure that this value is already set when building the solution DFA, so DFAs could be rejected already in this stage, if $\mathfrak{D}$ did not match.

We close this work with a short lookout.

During our requirements analysis we defined several parameters that have not been or only sparsely further discussed in here. This includes especially boundaries for the number of ingoing transitions to each state and drawing DFAs in a visual comprehensible manner. Connected to the latter is the question, whether a good procedure exists, that outputs a visual representation of a DFA via LaTeX-code, such that hand-made adjustments might be done afterwards. One could also think of making more parameters ranged, such that per instance a minimum and maximum number of states could be specified as input.

Regarding the planarity test as it is used now, one might ask whether there is a more efficient planarity test that is tailored to DFAs. Moreover it could be worth investigating whether informations generated during the planarity test can be used for drawing the DFA.

Our summary on research on DFA generation indicated that efficient - randomized and enumerating - methods to generate DFAs have already been found, whereas the resulting DFAs where even accessible. An improved version of the associated implementation could implement some of these methods or make use of existing implementations. We shall cite in this regard the enumeration method of Almeida et. al. [1] which uses a similar string representation of DFAs to iterate through all DFAs. Carayol and Nicaud [8] presented a randomization method that is deemed easy to implement.

Appendix A

An isomorphy test for DFAs

Here follows a simple isomorphy test that tries essentially to build a bijection as described in section 2.1.2.

```
function AreIsomorph ($A_1, A_2$)
    if $|Q_1| \neq |Q_2|$ or $|F_1| \neq |F_2|$ or $\Sigma_1 \neq \Sigma_2$ then
        return $false$
    $\pi(s_1) = s_2$                                   ▷ bijection $Q_1 \to Q_2$
    $O \leftarrow \emptyset$                           ▷ observed states
    $V \leftarrow \{s_1\}$                             ▷ visited states
    $q_c \leftarrow s_1$                               ▷ current state
    while true do
        for $((q_1, \sigma), p_1)$ in $\delta_1$ do    ▷ iterate through $d^+(q_c)$
            if $q_1 = q_c$ then
                continue
            $p_2 \leftarrow \delta_2(\phi(q_c), \sigma)$
            $p1marked \leftarrow (\pi(p_1) \neq \bot)$     ▷ see if $p_1$, $p_2$ were "marked" by $\pi$
            $p2marked \leftarrow (\exists q\colon \pi(q) = p2)$
            if $p1marked$ and $p2marked$ then
                if $\pi(p_1) \neq p_2$ then
                    return false
            else if $\neg p1marked$ and $\neg p2marked$ then
                $\pi(p_1) = p_2$
                if $p_1 \notin V$ then
                    Add $p_1$ to $O$
            else        ▷ one of $p_1$, $p_2$ was assigned to some state $\neq p_1$ resp. $p_2$
                return false
        if $|O| = 0$ then
            break
        Pick and remove $q_c$ from $O$
        Add $q_c$ to $V$
    end
    for $q_1$ in $F_1$ do
        if $\pi(q_1) \notin F_2$ then
            return false
    return true
```

This algorithm *visits* one by one all states of A_1 and tries to build π on the way. The currently visited state is denoted q_c. In $V \subseteq Q_1$ we save all already visited states. The set $O \subseteq Q_1$ shall contain all *observed* states, meaning those, that we encountered while following a transition, but have not been visited yet.

We call states of A_1, A_2 *marked*, if they have been assigned to another state by π. So if $\pi(q_1) = q_2$, then q_1, q_2 are marked. States in O will have the property, that they are marked.

Starting with $q_c = s_1$, in every while-iteration all outgoing transitions $\delta_1(q_c, \sigma) = p_1$ of the current state are followed. We then compute $\delta_2(\phi(q_c), \sigma) = p_2$, which is the state in A_2 that should correspond to p_1 of A_1. At this point (line 17) we do a case differentiation:

- p_1, p_2 both marked: Then we only need to ensure they are assigned to each other.

- p_1, p_2 both not marked: Then we can assign them to each other and may now add q_1 to O, since we observed it on an outgoing transition and know it has been marked. We will not add it, if we have visited as q_c.

- one of p_1, p_2 marked, one not: In that case they cannot be assigned to each other, since then both states would be marked; consequently one state has been assigned to a distinct third state.

When finished with visiting all outgoing transitions of a state q_c, we can pick the next state which is added to the visited states.

If all states of A_1 have been visited and the bijection thus been fully constructed, we need only to ensure, that the final state sets are equal after a renaming according to π.

Bibliography

[1] André Almeida, Marco Almeida, José Alves, Nelma Moreira, and Rogério Reis. Fado and guitar. volume 5642, pages 65–74, 07 2009. doi:10.1007/978-3-642-02979-0_10.

[2] Marco Almeida, Nelma Moreira, and Rogério Reis. Aspects of enumeration and generation with a string automata representation. *Theoretical Computer Science*, 387:93–102, 06 2009. doi:10.1016/j.tcs.2007.07.029.

[3] Frederique Bassino and Andrea Sportiello. Linear-time generation of specifiable combinatorial structures: general theory and first examples, 2013. arXiv:1307.1728.

[4] Frédérique Bassino, Julien David, and Cyril Nicaud. Regal: A library to randomly and exhaustively generate automata. 07 2007. doi:10.1007/978-3-540-76336-9_28.

[5] Frédérique Bassino, Julien David, and Andrea Sportiello. Asymptotic enumeration of minimal automata. *Leibniz International Proceedings in Informatics, LIPIcs*, 14, 09 2011. doi:10.4230/LIPIcs.STACS.2012.88.

[6] Frédérique Bassino and Cyril Nicaud. Enumeration and random generation of accessible automata. *Theoretical Computer Science*, 381:86–104, 08 2007. doi:10.1016/j.tcs.2007.04.001.

[7] Daniel Berend and Aryeh Kontorovich. The state complexity of random dfas. *Theoretical Computer Science*, 652, 07 2013. doi:10.1016/j.tcs.2016.09.012.

[8] Arnaud Carayol and Cyril Nicaud. Distribution of the number of accessible states in a random deterministic automaton. volume 14, pages 194–205, 02 2012. doi:10.4230/LIPIcs.STACS.2012.194.

[9] Jean-Marc Champarnaud and Thomas Paranthoën. Random generation of dfas. *Theoretical Computer Science*, 330:221–235, 02 2005. doi:10.1016/j.tcs.2004.03.072.

[10] Michael Domaratzki, Derek Kisman, and Jeffrey Shallit. On the number of distinct languages accepted by finite automata with n states. *J. Autom. Lang. Comb.*, 7(4):469–486, September 2002.

[11] Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Comb. Probab. Comput.*, 13(4–5):577–625, July 2004. URL: https://doi.org/10.1017/S0963548304006315, doi:10.1017/S0963548304006315.

[12] Gesellschaft für Informatik. Empfehlungen für bachelor- und masterprogramme im studienfach informatik an hochschulen (juli 2016), 2016. Accessed: 2020-02-12. URL: https://dl.gi.de/bitstream/handle/20.500.12116/2351/58-GI-Empfehlungen_Bachelor-Master-Informatik2016.pdf.

[13] John Hopcroft and Robert Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, October 1974.

[14] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[15] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - (2. ed.)*. Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001.

[16] Pierre-Cyrille Héam and Jean-Luc Joly. On the uniform random generation of determinisitic partially ordered automata using monte carlo techniques. 12 2014.

[17] Johanna Högberg and Lars Larsson. Dfa minimisation using the myhill-nerode theorem. Accessed: 2020-02-09. URL: `http://www8.cs.umu.se/kurser/TDBC92/VT06/final/1.pdf`.

[18] William Kocay. The hopcroft-tarjan planarity algorithm. October 1993.

[19] A. Korshunov. Enumeration of finite automata. *Problemy Kibernetiki*, page 34:5–82, 1959. In russian.

[20] C. Nicaud. *Étude du comportement en moyenne des automates finis et des langages rationnels*. PhD thesis, Université Paris 7, 2000.

[21] Cyril Nicaud. Random deterministic automata. pages 5–23, 08 2014. `doi:10.1007/978-3-662-44522-8_2`.

[22] Albert Nijenhuis and Herbert S. Wilf. *Combinatorial Algorithms*. Academic Press, 1978.

[23] Rogério Reis, Nelma Moreira, and Marco Almeida. On the representation of finite automata. *7th International Workshop on Descriptional Complexity of Formal Systems, DCFS 2005 - Proceedings*, 06 2005.

[24] Uwe Schöning. *Theoretische Informatik - kurzgefasst*. Spektrum, Akad. Verl, Heidelberg Berlin, 2001.

[25] V. Vyssotsky. A counting problem for finite automata. Technical report, 1959. Bell Telephon Laboratories.

[26] Thomas Schwentick Wim Martens. Theoretische informatik i ss18. Lecture notes, 2018.

Erklärung

Hiermit versichere ich, Gregor Hans Christian Sönnichsen, dass ich die vorliegende Arbeit selbständig verfasst habe, keine anderen als die von mir angegebenen Quellen und Hilfsmittel benutzt habe und die Arbeit nicht bereits zur Erlangung eines akademischen Grades eingereicht habe.

Bayreuth, den 8. Februar 2020.

Gregor Hans Christian Sönnichsen