UNIVERSITÄT
BAYREUTH

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

Bachelorarbeit

Automatic Generation of DFA Minimization Problems

Automatische Generierung von DFA Minimierungsproblemen

Gregor Hans Christian Sönnichsen

July 3, 2020

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

Abstract

The theory of deterministic finite automata (DFAs) is a classical topic of computer science-related courses. A typical problem to solve for students is: Given a task DFA A_{task}, minimize it. Generation of such problems is often done manually by the exercise instructor. This work presents ideas to automize the generation of DFA minimization problems.

We start in chapter 2 with introducing some preliminaries, in particular Hopcroft's minimization method. Based on that algorithm, we will deduce requirements, parameters and structure for a generator of DFA minimization problems. The structure will be the following: Firstly generate the solution DFA, then create equivalent state pairs and lastly add unreachable states. This approach is, intuitively spoken, Hopcroft's algorithm in reverse.

Concerning the generation of solution DFAs (chapter 3) we make use of a simple rejection algorithm, that generates test DFAs by randomization or enumeration and rejects them, if they do not match the demanded properties. From results in related research some conclusions will be drawn for this work.

In chapter 4 we describe the extension of a solution DFA 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 of equivalent state pairs and unreachable states.

Zusammenfassung

Die Theorie endlicher deterministischer Automaten (DEAs) ist ein klassisches Thema der Lehre mit Informatikbezug. Eine typisches Problem das Studenten hier lösen sollen ist: Gegeben einen Aufgaben-DEA A_{task}, minimiere ihn. Das Generieren solcher Minimierungsprobleme wird allerdings häufig manuell vom Übungsleiter vorgenommen. In dieser Arbeit werden somit Ideen präsentiert um DEA Minimierungsprobleme automatisiert zu generieren.

Wir beginnen in Kapitel 2 mit einigen einführenden Definitionen, darunter insbesondere Hopcroft's Minimierungsalgorithmus. Auf dessen Basis werden wir Voraussetzungen, Parameter und Struktur eines Programms erarbeiten, das DFA Minimierungsprobleme generiert. Die Struktur dieses Programms wird wie folgt sein: Zunächst wird der Lösungs-DEA generiert, dann werden äquivalente Zustandspaare erzeugt und schließlich unerreichbare Zustände hinzugefügt. Diesen Ansatz kann man grob als Umkehrung von Hopcroft's Algorithmus charakterisieren.

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. Aus relevante Ergebnissen verwandter Forschungsarbeit werden wir einige Schlussfolgerungen für diese Arbeit ziehen können.

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 [13, pp. 5-6]. It provides the chance to gain a precise cognitive model of a theory, possibly yielding new perspectives on other problems and topics. 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. We 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 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 ([21, pp. 1-4]):

- freeing time for other things, e.g. research, helping students face-to-face, designing exercise sheets
- increased predictability and consistency of the generated task properties, which can be adjusted accurately through various parameters
- saves humans from possibly monotonous work

Engagement on this topic promises moreover increased clarification which kind of minimization tasks can be generated, and where difficulties of such 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, which 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 $\{0, \ldots, n-1\}$.

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 $q \in Q$, $w \in \Sigma^*$, $\sigma \in \Sigma$

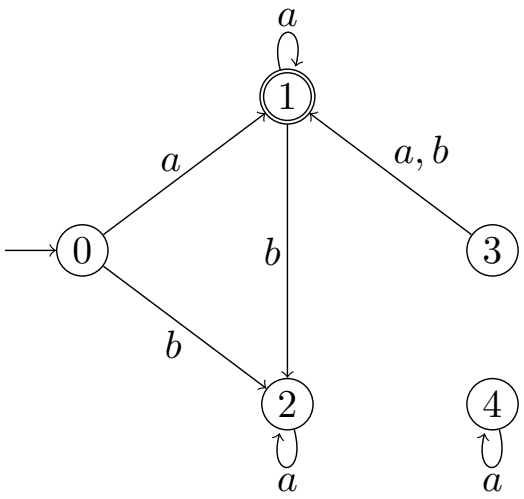
Then, the *language* of A is defined as $L(A) = \{\ w \mid \delta^*(w) \in F\ \}$ [15, pp. 49-50. 52].

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 [10, 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* [10, 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. From now on we will only work with complete DFAs.



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

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

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 $\delta(4, b)$ are not defined.

Definition 2 (Minimal DFA)**.** We call a DFA A *minimal*, if there exists no other DFA with the same language having 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, where

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

If $q_0 \not\sim_A q_1$, then q_0 and q_1 are called a *distinguishable* state pair. It is well-known (see for instance [15, p. 160]) that 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 *isomorphic*, 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 [27, 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 appendix A.

2.1.3 Hopcroft's Minimization Algorithm

There are three major algorithms for DFA minimization found by Moore, Brzozowski and Hopcroft, respectively ([8, p. 2]). On the latter an easy algorithm is based, that is presented in the textbook [15] and will be described here more precisely. We will call Hopcroft's algorithm 'the minimization algorithm' from now on. Its general structure is the following:

1: **function** MinimizeDFA(A)
2: $A' \leftarrow$ RemUnreachables(A, FindUnreachables(A))
3: **return** RemEquivPairs(A', FindEquivPairs(A'))

It can be seen that MinimizeDFA works in four major steps. In step 1 and 2 unreachable states are found and removed. In step 3 all equivalent state pairs are found. Step 4 removes states in such a way, that no equivalent state pairs are left.

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

1: **function** FindUnreachables(A)
2: $U \leftarrow Q \setminus \{s\}$ ▷ undiscovered states
3: $O \leftarrow \{s\}$ ▷ observed states
4: $D \leftarrow \{\}$ ▷ discovered states
5: **while** $|O| > 0$ **do**
6: $N \leftarrow \{\ p \mid \exists q \in O\colon\ \sigma \in \Sigma\colon\ \delta(q,\sigma) = p\ \wedge\ p \notin O \cup D\ \}$
7: $U \leftarrow U \setminus N$
8: $D \leftarrow D \cup O$
9: $O \leftarrow N$
10: **return** U

2. Remove all unreachable states and their transitions.

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

3. Compute all equivalent state pairs ($\sim_A$). The representation is inspired by Martens and Schwentick [22, ch. 4, p. 17]. Note that FindEquivPairs requires its input automaton to be complete ([8, p. 13]).

1: **function** FindEquivPairs(A)
2: $M \leftarrow \{(p,q),(q,p) \mid p \in F, q \notin F\}$
3: **do**
4: $M' \leftarrow \{(p,q) \mid (p,q) \notin M \wedge \exists\sigma \in \Sigma\colon (\delta(p,\sigma),\delta(q,\sigma)) \in M\}$
5: $M \leftarrow M \cup M'$
6: **while** $M' \neq \emptyset$
7: **return** $Q^2 \setminus M$

4. Merge all equivalent state pairs, which are exactly those in $\sim_A$. The representation is inspired by Högberg and Larsson [18, p. 10].

1: **function** RemEquivPairs(A, $\sim_A$)
2: $Q' \leftarrow \emptyset, \delta' \leftarrow \emptyset, F' \leftarrow \emptyset$
3: **for** q **in** Q **do**
4: Add $[q]$ to Q' ▷ $[\cdot]_{\sim_A}$ shall be abbreviated $[\cdot]$
5: **for** σ **in** Σ **do**
6: $\delta'([q],\sigma) = [\delta(q,\sigma)]$
7: **if** $q \in F$ **then**
8: Add $[q]$ to F'
9: **return** $(Q', \Sigma, \delta', [s], F')$

The following theorem states the most important property of MinimizeDFA.

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

When looking at FindEquivPairs, 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 $m\colon \mathbb{N} \to \mathcal{P}(Q \times Q)$:

1: **function** m-FindEquivPairs(A)
2: $i \leftarrow 0$
3: $m(0) \leftarrow \{(p,q),(q,p) \mid p \in F, q \notin F\}$
4: **do**
5: $i \leftarrow i + 1$
6: $m(i) \leftarrow \{(p,q),(q,p) \mid (p,q) \notin \bigcup m(\cdot) \wedge \exists \sigma \in \Sigma\colon (\delta(p,\sigma),\delta(q,\sigma)) \in m(i-1)\}$
7: **while** $m(i) \neq \emptyset$
8: **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 of FindEquivPairs in exchange.

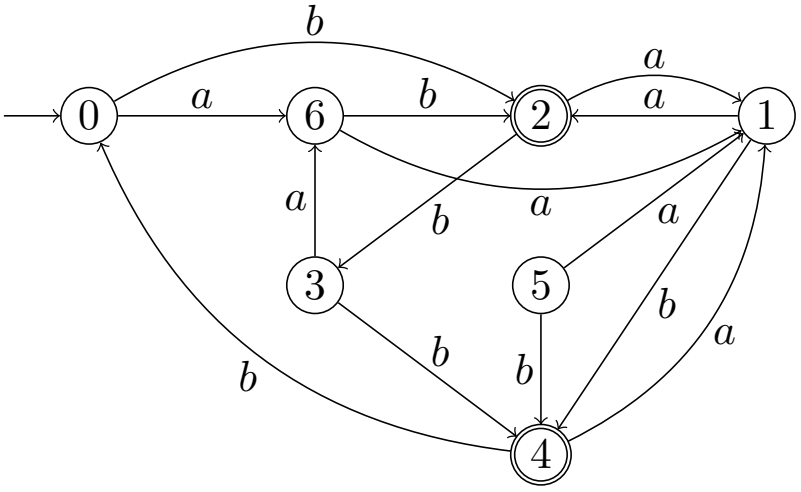
Definition 4. We denote the number of iterations done by FindEquivPairs on a DFA A as $\mathfrak{D}(A)$.

This number will prove to be a major characteristic of difficulty for a minimization task.

2.2 DFA Minimization Problems

Now that we have introduced all necessary basic definitions, we present an example DFA minimization task and its sample solution, as it could have been given to students in an introductory course on automata theory. In this context further naming conventions will be determined.

Task: Consider the below shown deterministic finite automaton A:



Apply the minimization algorithm and illustrate for each state pair of A during which FindEquivPairs*-iteration it was marked. Draw the resulting automaton.*

Figure 2.2: An example DFA minimization task.

Figures 2.2 and 2.3 show a DFA minimization task and its solution. The students are confronted with a *task DFA* A_{task}, which has to be minimized — we will assume that Hopcroft's algorithm is used.

As a consequence (following alg. 0) firstly, unreachable states have to be eliminated, we then gain A_{re} (having only *reachable* states). Secondly equivalent state pairs of A_{re}

Solution:
Step 1: Detect and eliminate unreachable states.

State 5 is unreachable.

Step 2: Apply FindEquivPairs *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	■	■	■	■	■	■

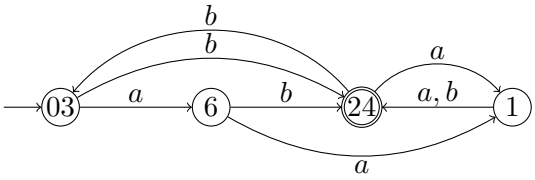


Figure 2.3: Solution to the DFA minimization task in fig. 2.2.

are merged such that the minimal *solution DFA* A_{sol} is found. The table T displayed in Figure 2.3 is nothing else but a visualization of the function m of m-FindEquivPairs, where $T(q_0, q_1) = i \Leftrightarrow (q_0, q_1) \in m(i)$.

Notice that step 2 and in particular the computation of $\sim_A$ can be seen as central: Step 1 is relatively easy, unreachable states can normally be detected visually without running a search algorithm. Moreover algorithms like breadth-first search are often introduced earlier in computer science education. In step 2 computing $\sim_A$ is conceptually harder to grasp than collapsing A_{re} towards the solution DFA.

Let us now formally define DFA minimization problems.

Definition 5 (DFAMinimization)**.**
Given: A DFA A_{task}*.*

Task: Compute $A_{sol} =$ MinimizeDFA(A_{task})*.*

Our goal in this thesis is to automatically generate instances of this problem. Before starting to think about concrete generation algorithms, we will devise some parameters and requirements to influence the properties of the generated problem.

2.3 Requirements and Parameters for a Generation Algorithm

Of some simple requirements we already know in advance. Firstly, we can state that A_{sol} has to be minimal regarding A_{re} and A_{task}. Secondly the languages of A_{task}, A_{re} and A_{sol} must be the same. Lastly we know that every state of A_{re} needs to be reachable.

Besides some other sensible requirements we will devise especially parameters for generation algorithms in this section. These parameters are going to be adjustment possibilities, which will in almost all cases allow to influence the difficulty of the generated problem in some way.

Since the difficulty of a DFA minimization task A_{task} for students is mainly determined by the difficulty of executing MinimizeDFA(A_{task}), most parameters will influence certain aspects of the execution of MinimizeDFA.

FindEquivPairs-iterations (d). Consider the computation of the sets $m(i)$ in m-FindEquivPairs (see alg. 0). Determining $m(0)$ is quite straightforward, because it

consists simply of tests whether two states are in $F \times Q \setminus F$ (see 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 line 6). Determining $m(2), m(3), \ldots$ requires the same rule. It shows nonetheless a students understanding of FindEquivPairs' terminating behavior: The algorithm does not stop after computing $m(1)$, but only when no more distinguishable state pairs were found.

It would therefore be sensible if $\mathfrak{D}(A_{task})$ could be adjusted by some parameter d.

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: $n_s, n_e, n_u \in \mathbb{N}$, where

- n_s is the number of states of the *solution* DFA A_{sol}
- n_e is the number of distinct *equivalent* state pairs of A_{re}
- n_u is the number of *unreachable* states of A_{task}

They can be equivalently described 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 FindEquivPairs.

Alphabet size (k). The more symbols the alphabet of A_{task}, A_{re} and A_{sol} has (note that 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 (see line 6 of alg. 0). In addition we will see later, that there exists no DFA with $k < 2$ having any equivalent state pairs, so $k \geq 2$ is sensible and even required if $n_e > 0$.

Number of final states (n_F). Most DFAs in teaching have about 1 to 3 final states (see e.g. [15, pp. 48-78] or [27, pp. 28-48]), so being able to set a number of final states allows concentrating on or deviating from familiar DFAs.

In this work n_F shall determine the number of final states in A_{sol}. Consequently, A_{task} may have minimum n_F and maximum $n_F + n_e + n_u$ final states. The latter is the case, if all equivalent states are created from final states and if every unreachable state is created as final state.

Uniqueness of solution DFA language. For example for an exam it would be sensible to be able to generate a task where the DFA language is unique, meaning there was no previously generated DFA with the same language.

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} always have the same language.

Completeness of task DFA (c). In opposition to FindEquivPairs, FindUnreachables and RemUnreachables do not require their input DFA A_{task} to be complete. So we could have unreachable states in A_{task}, to which δ is not defined for all alphabet symbols. It is however sensible, to build task DFAs complete too to avoid possible confusion: Such subtleties do not highlight the main ideas of MinimizeDFA.

Nonetheless we shall introduce a boolean parameter c, that determines if there exist unreachable states, that make A_{task} incomplete. Thus an exercise lecturer has the option, to showcase this matter on a DFA and generate according exercises.

Planar drawing of task DFA (p_{sol}, p_{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 and its associated implementation libraries and parameters $p_{sol}, p_{task} \in \{0, 1\}$ (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 task DFA. 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. Here the inclusion of a maximum degree parameter is omitted.

2.4 Approach and General Algorithm

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 together will fulfill all criteria chosen above and are covered each in depth in chapter 3 respectively chapter 4.

At the beginning of chapter 3 and 4, 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:

1: **function** GenDFAMinProblem($n_s, k, n_F, d, p_{sol}, p_{task}, n_e, n_u, c$)
2: $A_{sol} \leftarrow$ GenNewMinDFA(n_s, k, n_F, d, p_{sol})
3: $A_{task} \leftarrow$ ExtendMinDFA($A_{sol}, p_{task}, n_e, n_u, c$)
4: **return** A_{task}

One might notice, that we pass d only to GenNewMinDFA, even though we want to ensure $d = \mathfrak{D}(A_{task})$ too. This is a valid approach due to the fact, that creating equivalent states and adding unreachable states does not change the $\mathfrak{D}$-value, which we are going to prove.

However since unreachable states are eliminated before FindEquivPairs is applied, we need only to prove, that creating equivalent states does not change the $\mathfrak{D}$-value, which will be done during the discussion of step 2, more specifically in section 4.1.5.

Chapter 3

Generating Minimal DFAs

We seek algorithms for generation of minimal DFAs that fulfill the conditions defined in the requirements analysis section 2.3. We formally subsume these conditions via the GenerateNewMinimalDFA-problem:

Definition 6 (GenNewMinDFA)**.**
Given:

$$\begin{aligned} n_s &\in \mathbb{N} && \textit{number of states} \\ k &\in \mathbb{N} && \textit{alphabet size} \\ n_F &\in \mathbb{N} && \textit{number of final states} \\ d &\in \mathbb{N} && \textit{value of } \mathfrak{D}(A_{sol}) \\ p &\in \{0,1\} && \textit{planarity-bit} \end{aligned}$$

Task: Compute, if it exists, a solution DFA A_{sol} with

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

To solve this problem we present one approach in much detail. Afterwards an alternative approach and related work will be discussed.

Remark. 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 DFAs with the properties determined by the input parameters. 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 has not been generated before. 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 did not restrict ourselves to $Q_{sol} = [[n_s]]$ and $\Sigma_{sol} = [[k]]$, then for a given number of states n_s and symbols k, 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 even guarantee now that the enumerating variant of our algorithm is going to terminate.

Here follows a description of our general algorithm for generating minimal DFAs.

```
1:  function GenNewMinDFA-1 (n_s, k, n_F, d ∈ ℕ, p ∈ {0,1})
2:      while True do
3:          generate DFA A_test with |Q|, |Σ|, |F| matching n_s, k, n_F
4:          if A_test not minimal or d ≠ 𝔇(A_test) then
5:              continue
6:          if p = 1 and A_test is not planar then
7:              continue
8:          if L(A_test) is not new then
9:              continue
10:         return A_test
```

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

3.1.1 Ensuring $\mathfrak{D}$-value lies in range and Minimality

In order to test, whether A_{test} is minimal, it is sufficient to ensure, that A_{test} has no equivalent state pairs and no unreachable states.

To get $\mathfrak{D}(A_{test})$, we have to run FindEquivPairs (or a variant) entirely. Hence we can combine the test for the existence of equivalent state pairs with computing the DFAs $\mathfrak{D}$-value:

```
1:  function HasEquivalentStates(A)
2:      depth ← 0                                   ▷ will be 𝔇(A) in the end
3:      M ← {(p,q), (q,p) | p ∈ F, q ∉ F}
4:      do
5:          depth ← depth + 1
6:          M' ← {(p,q) | (p,q) ∉ M ∧ ∃σ ∈ Σ: (δ(p,σ), δ(q,σ)) ∈ M}
7:          M ← M ∪ M'
8:      while M' ≠ ∅
9:      hasDupl ← |{(p,q) | p ≠ q ∧ (p,q) ∉ M}| > 0
10:     return hasDupl, depth
```

Since FindEquivPairs 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 FindUnreachables and test whether the computed set is empty:

```
1:  function HasUnreachableStates(A)
2:      return |FindUnreachables(A)| > 0
```

3.1.2 Ensuring Planarity

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 information on this can be found for example in this [19] introduction from William Kocay. The original paper describing the algorithm is by Hopcroft and Tarjan [16].

3.1.3 Ensuring Uniqueness

In our requirements we stated, that we wanted the generated solution DFA to be new, meaning it should have a unique language compared to all previously generated solution DFAs. This implies the need of a database, such that we can save and load the list l of already found DFAs. We name this database $\mathrm{DB}_{\mathrm{found}}$.

In DB_{found} only minimal DFAs will be saved, since every solution DFA is minimal. Furthermore every test DFA A_{test} is guaranteed to be minimal, since non-minimal test DFAs are filtered out beforehand.

Theorem 1 states that two minimal DFAs have the same language, iff they are isomorphic. So we can use an isomorphism test to decide whether two DFAs have the same language.

However this test can be constructed more efficiently: Two minimal DFAs cannot be isomorphic, if their number of states, alphabet size or number of final states differ. It would thus suffice to compare the test DFA only against those DFAs from the database that match the parameters n_s, k, n_F. As a consequence we propose the following scheme for DB_{found}, which is assumed to be a relational database:

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

Now we need not fetch all DFAs every time but can select only the relevant ones.

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

```
1: function GenNewMinDFA-2 (n_s, k, n_F, d, p)
2:     l ← all DFAs in DB_found matching n_s, k, n_F
3:     while True do
4:         ...
5:         if A_test is isomorphic to any DFA in l then
6:             continue
7:         save A_test and its respective properties in DB_found
8:         return A_test
```

A sample isomorphism test is described in Appendix A.

3.1.4 Option 1: Generating Test DFAs via Randomness

We now approach the task of generating a random DFA where 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.

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

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.

```
1: function GenNewMinDFA-3a (n_s, k, n_F, d, p)
2:     l ← all DFAs in DB1 matching n_s, k, n_F
3:     Q ← [[n_s]]
4:     Σ ← [[k]]
5:     while True do
6:         δ ← ∅
7:         for (q, σ) in Q × Σ do
8:             δ(q, σ) = random chosen state from Q
9:         s ← 0
10:        F ← random sample of n_F states from Q
11:        A_test ← (Q, Σ, δ, s, F)
12:        if A_test not minimal or d ≠ 𝔇(A_test) then
13:            continue
14:        if p = 1 and A_test is not planar then
15:            continue
16:        if A_test is isomorphic to any DFA in l then
17:            continue
18:        save A_test and its respective properties in DB1
19:        return A_test
```

3.1.5 Option 2: Generating Test DFAs via Enumeration

The second method of test DFA generation is based on the idea, that instead of randomly generating F and δ, we could enumerate through all possible final state sets and transition functions. We will use the given parameters n_s, k, n_F to split up the enumeration space — every enumeration yields only DFAs with the same n_s, k, n_F.

An enumeration is represented by an *enumeration state* $s_{n_s,k,n_F} = (F_F, F_\delta)$ consisting of two arrays[2]. The first array shall have n_s Bits, where Bit $F_F[i] \in \{0, 1\}$ represents the information, whether i is a final state or not. The second array 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:*

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

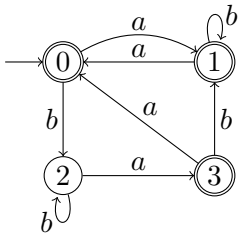
A possible F_δ-array could be 12013201. *The following table depicts, how F_δ assigns one state to every combination of states and symbols $(q, \sigma) \in Q \times \Sigma$ and thus defines δ:*

$q \in Q$	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

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

[2] We denote the two arrays (or *fields*) as F_F, F_δ instead of A_F, A_δ to avoid confusion with the notation of DFAs.

The corresponding DFA might look like this:



□

Given an enumeration state $s_{n_s,k,n_F} = (F_F, F_\delta)$ we will then compute the next state representing a DFA using the following algorithm. We assume, that adding 1 to a n-ary number $n-1\ n-1\ \ldots n-1$ yields $00\ldots0$.

1: **function** IncrementEnumProgress $(F_F, F_\delta, n_s, k, n_F)$
2: add 1 to $(F_\delta)_{n_s}$
3: **if** $F_\delta = 0\ldots0$ **then**
4: **if** $F_F = 1\ldots10\ldots0$ **and** $\#_1(F_F) = n_F$ **then**
5: **return** $\perp$
6: **while** $\#_1(F_F) \neq n_F$ **do**
7: add 1 to $(F_F)_2$
8: **return** F_F, F_δ

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 all transitions functions have been enumerated (F_δ becomes $0\ldots0$), then we increment F_F until it contains n_F 1's (again) and enumerate all transition functions again.

Concerning the terminating behavior of IncrementEnumProgress consider the following observation:

Observation. *An enumeration of DFAs given n_s, k, n_F will eventually terminate. 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.*

This leads us to the range of an enumeration which can be expressed using the first and last enumeration state:

- $s_{n_s,k,n_F} = (F_F, F_\delta) = (0\ldots0\underbrace{1\ldots1}_{n_F\ 1's},\ 0\ldots0)$
- $s_{n_s,k,n_F} = (F_F, F_\delta) = (\underbrace{1\ldots1}_{n_F\ 1's}0\ldots0,\ n_s-1\ldots n_s-1)$

Consequently the algorithm can terminate in two ways. Either a next DFA could be enumerated or all DFAs with $|Q| = n_s, |\Sigma| = k, |F| = n_F$ have been found.

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 $a, b, \ldots$ instead of $0, 1, \ldots$ as symbols. Valid enumeration progresses are depicted green.*

We will start with the initial enumeration progress (1). 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. Thus the next enumeration step yields the state (7), that marks the enumeration end.*

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

□

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

```
1:  function DFAfromEnumProgress (F_F, F_δ, n_s, k, n_F)
2:      Q ← [[n_s]]
3:      Σ ← [[k]]
4:      δ ← ∅
5:      for i in [[n_s]] do
6:          for j in [[k]] do
7:              δ(i, j) = F_δ[i * k + j]
8:      s ← 0
9:      for i in [[n_s]] do
10:         if F_F[i] = 1 then
11:             Add i to F
12:     return (Q, Σ, δ, s, F)
```

Since it is not sensible to create initial enumeration states for all possible enumeration (there are infinitely many, due to $n_s \in \mathbb{N}$) we will create enumeration states by demand.

Once the enumeration to the next DFA within a call of GenNewMinimalDFA has been finished, it is reasonable to save the enumeration state, such that during the next call enumeration can be resumed from that point on. We will run a different enumeration for every parameter combination. Thus we introduce a second database DB_{states} with the following table:

$$|Q_A| \quad |\Sigma_A| \quad F_F \quad F_\delta \quad d_{min} \quad d_{max} \quad planar$$

In the following variant of GenNewMinimalDFA it can be seen how the enumeration method is integrated:

```
1:  function GenNewMinDFA-3b (n_s, k, n_F, d, p)
2:      l ← all DFAs in DB_found matching n_s, k, n_F, d, p
3:      if ∃ enum. state for n_s, k, n_F in DB_states then
4:          F_F, F_δ ← load s_{n_s,k,n_F} from DB_states
5:      else
6:          F_F, F_δ ← 0...0 1...1 0...0   (n_F 1's)
```

```
7:  while True do
8:      if F_F, F_δ is finished then
9:          save F_F, F_δ
10:         return ⊥
11:     A_test ← next DFA based on F_F, F_δ
12:     if A_test not minimal or d ≠ 𝔇(A_test) then
13:         continue
14:     if p = 1 and A_test is not planar then
15:         continue
16:     if A_test is isomorphic to any DFA in l then
17:         continue
18:     save F_F, F_δ in DB_states
19:     save A_test and its respective properties in DB_found
20:     return A_test
```

3.2 Related Work on DFA Generation

Nicaud provides an overview of results on random generation and combinatorial properties of DFAs in [24]. We will outline relevant related work.

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 [10] continue ideas started by Nicaud in his thesis [23]. 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 the properties $v \in N \Rightarrow d^+(v) = m$ and $v \in L \Rightarrow d^+(v) = 0$ (essentially nodes and leaves). 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 [10, 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[3]" 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 [10, p. 8]).

As a consequence Champarnaud and Paranthoën are able to construct a random generator of accessible complete DFAs using the "recursive method" from [25] which generates n-tuples [10, 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 [24, pp. 10-11].

Almeida et al. [1, 2, 26] present and implement methods using a string-encoding of DFAs for exact enumeration and random generation of DFAs. Nicaud [24, p. 11] states in a remark, that this approach uses the same recursive method and differs only in the DFA encoding.

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

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 sizes of its generated objects are not fixed but in an ε-interval around a given input size - this stands in opposition to most random generators in literature [12, 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 $\mathcal{O}(n^{1.5})$ for a single random generation.

Using a Rejection Algorithm. Carayol and Nicaud [9] give a simple algorithm with the same runtime complexity ($\mathcal{O}(n^{1.5})$). 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 [9]. The enumeration method uses encodings of DFAs similar to those used by Almeida et al. [26].

3.3 Empirical and Combinatorial Results

Concerning combinatorial properties of DFAs, several authors (e.g. [6, 11, 17]) consider a work from Vyssotsky [28] in the Bell laboratories to be the first on this subject. A contribution by Korshunov [20] 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].

More recent 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 [11] give some asymptotic estimates and explicit computations for the number of several types of distinct languages and automata. The here relevant results have been subsumed and extended in [2, p. 8] and were empirically confirmed in [4].

In Figure 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 (assuming $s = 0$, $Q = [[n]], \Sigma = [[k]]$) 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 random 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. The numbers furthermore show that for a "sensible" choice of parameters (e.g. $n \geq 3$, $k \geq 2$) there are more than enough distinct minimal DFAs that can be generated.

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

$\lvert\Sigma\rvert$ (k)	$\lvert Q\rvert$ (n)	$\lvert\mathcal{A}_{min,n,k}\rvert$	$\lvert\mathcal{A}_{n,k}\rvert$	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 exact amount of distinct 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].

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 sec. 2.3):

Definition 7 (ExtendMinDFA)**.**

Given:

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

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*[1]
- $u_1, \ldots, u_{n_u}$ *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 isomorphic to* MinimizeDFA(A_{task})

In order to fulfill these requirements we will deduce for equivalent and unreachable states how they may be added by examining their desired properties. If we get to have the choice between several equally good options, we will randomly select one.

We will be guided by the separation of creating equivalent and unreachable states as in Hopcrofts algorithm, thus distinct algorithms will be devised to create each kind of states. Concerning the planarity option, we again use a rejection algorithm with the same test as before (see sec. 3.1.2). These decisions lead to the following formulation of ExtendMinDFA:

[1] The states $r_1, \ldots, r_{n_e}$ can be seen as being *redundant*.

```
1: function ExtendMinDFA(A_sol, n_e, n_u, p, c)
2:     if p = 0 then
3:         A_re ← AddUnrStates(A_sol, n_u, c)
4:         return CreateEquivPairs(A_re, n_e)
5:     else
6:         A_task ← ⊥
7:         while A_task not planar do
8:             A_re ← AddUnrStates(A_sol, n_u, c)
9:             A_task ← CreateEquivPairs(A_re, n_e)
10:        return A_task
```

We will show for the action of adding equivalent states, that this does not influence a DFAs $\mathfrak{D}$-value — thus we do not have to care whether we are possibly changing it.

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 the added states is equivalent to a state in Q_{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.

In our algorithm we will add each $r = r_i,\ i \in [1, n_e]$ separately to A_{sol}. Consider the properties $r_1, \ldots, r_{n_e}$ must have. Since we start from A_{sol}, and add in each step a state, that will be equivalent to a state in the so-far constructed DFA, it follows by transitivity, that each r will be equivalent to an *origin* state e of A_{sol}.

$$\forall r \in \{r_1, \ldots, r_{n_e}\}\colon\ \exists e \in Q_{sol}\colon\ r \sim_A e$$

In our algorithm, we will first choose a to-be-equivalent-state $e \in Q_{sol}$ for each state r we create, then we add its transitions.

4.1.1 Adding Outgoing Transitions

Regarding the outgoing transitions of any r equivalent to a state e, we are directly restricted by their equivalence relation:

$$\begin{aligned}
& r \sim_A e \\
\Rightarrow\ & \forall z \in \Sigma^*\colon\ (\delta^*(r, z) \in F \Leftrightarrow \delta^*(e, z) \in F) \\
\Rightarrow\ & \forall \sigma \in \Sigma\colon \\
& \qquad \delta(r, \sigma) = q_1 \wedge \delta(e, \sigma) = q_2 \wedge \\
& \qquad \forall z' \in \Sigma^*\colon (\delta^*(q_1, z') \in F \Leftrightarrow \delta^*(q_2, z') \in F) \\
\Rightarrow\ & \forall \sigma \in \Sigma\colon \\
& \qquad \delta(r, \sigma) = q_1 \wedge \delta(e, \sigma) = q_2 \wedge q_1 \sim_A q_2 \\
\Rightarrow\ & \forall \sigma \in \Sigma\colon [\delta(r, \sigma)]_{\sim_A} = [\delta(e, \sigma)]_{\sim_A}
\end{aligned}$$

So we see, that the state $\delta(r, \sigma) = q$ must be in the same equivalence class as $\delta(e, \sigma) = p$. We may thus formulate the rule for adding outgoing transitions to a new state quite straightforwardly:

R1: For each symbol $\sigma \in \Sigma$ choose exactly one state (A shall be complete) $q \in [\delta(e,\sigma)]_{\sim_A}$ and set $\delta(r,\sigma) = q$.

Since the solution DFA is complete and since every here added state gets a transition for every alphabet symbol, we know that every $[\delta(e,\sigma)]_{\sim_A} \neq \emptyset$, so the rule is guaranteed to be fulfillable.

Note that this substep does not change the equivalence class of any other now existing state, since r cannot be reached yet (and thus it can not lie on a path from any other state to a final state) - it has no ingoing transitions.

4.1.2 Adding Ingoing Transitions

First of all, we know, that r must be reachable, since we decided that all unreachable states will be added later. So we need to give r at least one ingoing transition. Doing this, we have to ensure, that any state q, that gets an outgoing transition to r remains in its equivalence class.

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

We see here, that p must have at least one "free" transition, else it would become unreachable. A state has a free transition, if it has at least two ingoing transitions or if it is start state and has at least one ingoing transition. With $in(q)$ we denote the *ingoing elements of* q:

$$in(q) = |d^-(q)| + \begin{cases} 1 & \text{if } s = q1 \\ 0 & \text{else} \end{cases}$$

Now we can define our rule for adding ingoing transitions:

R2: Choose at least one $((q,\sigma),p) \in \delta$ with $[p] = [e]$ and $in(p) \geq 2$. Remove $((q,\sigma),p)$ from δ and add $((q,\sigma),r)$.

4.1.3 Requirements for choosing origin states

From rule **R2** we know that the equivalence class of any origin state e has to contain at least one state with at least two ingoing elements. We establish the following notion to pin down the requirement which all origin states have to fulfill when they are chosen:

$$duplicatable(q) \Leftrightarrow_{def} (\exists p \in [q]_{\sim_A} : in(p) \geq 2)$$

Observation. *The number of duplicatable states in any accessible DFA A is*

$= 0$, *if* $|\Sigma| \leq 1$ *and* $|d^-(s)| = 0$*: The start state has no free ingoing transition and neither have all other states.*

$= 1$, *if* $|\Sigma| \leq 1$ *and* $|d^-(s)| > 0$*: The start state has* $|d^-(s)| - 1$ *free ingoing transitions, but other states still do not have enough.*

> 1, *if* $|\Sigma| > 1$*, due to the pigeonhole principle: An accessible complete DFA has* $|Q||\Sigma|$ *transitions which have to be spread across* $|Q|$ *states.*

This observation guarantees, that there exists a duplicatable state, if $k > 1$ or $k = 1$ and $|d^-(s)| > 0$. Note that we do not guarantee, that n_e equivalent states can be added.

Lemma 1. *Adding the outgoing transitions of r does not change whether its associated state e is duplicatable.*

Proof. Case 1: e is duplicatable. Adding transitions to the DFA will never change that, since $|d^-(e)|$ will stay the same or increase, but never decrease.

Case 2: e is not duplicatable. Towards a contradiction: The only outgoing transition from r, that could make e duplicatable, would be $\delta(r, \sigma) = e$.

Adding this transition by rule **R1** would require, that there exists a transition $\delta(e, \sigma) \in [e]$ (see **R1**).

By the definition of $\sim_A$, every state in $[e]$ then has to have an outgoing transition to a state in $[e]$.

Since e is not duplicatable, all states in $[e]$ have at maximum one ingoing elements. If $s \in [e]$, then there are not enough ingoing transitions to match all these outgoing transitions.

Else the required $|[e]|$ outgoing transitions would form all ingoing transitions of the $|[e]|$ states in $[e]$.

But then there would be no ingoing element left that ensures the reachability of $[e]$. □

4.1.4 The Algorithm

We integrate our strategy of creating equivalent state pairs in the following algorithm. It basically consists of a for-loop that adds one state per iteration. In each iteration first a duplicatable state in the so-far built DFA is determined. Then the new state is created and outgoing and ingoing transitions are added.

function CreateEquivPairs(A, n_e)
  if $k = 0$ **or** ($k = 1$ **and** $|d^-(s)| = 0$) **then**
    return $\perp$
  $Q \leftarrow Q_{sol}$
  $\delta \leftarrow \delta_{sol}$
  $F \leftarrow F_{sol}$
  $\sim_A \leftarrow Q \times Q$
  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 $[q]_{\sim_A}$
        break
    if $e = \perp$ **then return** $\perp$
    $r \leftarrow$ unused state label ▷ create to e equivalent state r
    Add r to Q
    Add $(r, r), (e, r), (r, e)$ to $\sim_A$
    for σ **in** Σ **do** ▷ **R1:** add $d^+(r)$
      $\delta(r, \sigma) =$ random chosen state from $[\delta(e, \sigma)]_{\sim_A}$
    $P \leftarrow \{\ ((s, \sigma), t) \in \delta \mid t \in [e]_{\sim_A},\ in(t) \geq 2\ \}$ ▷ **R2:** add $d^-(r)$
    $C \leftarrow$ random nonempty subset of P
    for $((s, \sigma), t)$ **in** C **do**
      $\delta(s, \sigma) = r$
  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.

4.1.5 Creating Equivalent State Pairs does not change $\mathfrak{D}$

In this section we want to prove that our method of creating state pairs does not affect the number of FindEquivPairs-iterations. Using this information we can be sure that $\mathfrak{D}(A_{sol}) = \mathfrak{D}(A_{task})$ and our just explained algorithm does not have to care about possibly changing this value.

To do this proof, we will first introduce two auxiliary definitions and then prove two minor lemmas. As a side effect, Lemma 2 will describe a central property of FindEquivPairs and Lemma 3 will show an extended characterization of $\mathfrak{D}(A)$ compared to its definition (def. 4).

A word w shall be called *finishing word of* q, iff $\delta^*(q, w) \in F$.

With $f(q) = \{\ w \mid \delta^*(q, w) \in F\ \}$ we denote the set of all finishing words to a state.

Definition 8. We will call a word w *distinguishing word of* p, q, iff $d_A(w, p, q)$ is true where

$$\begin{aligned} d_A(w,p,q) \text{ is true} &\Leftrightarrow (\delta^*(p,w) \in F \Leftrightarrow \delta^*(q,w) \notin F) \\ &\Leftrightarrow (w \in f(p) \Leftrightarrow w \notin f(q)) \end{aligned}$$

This definition and its terminology are in close relation to definition 3. The following lemma and its proof are in parts inspired by Martens and Schwentick [22, ch. 4 p. 18].

Lemma 2. *In the context of* FindEquivPairs *the following is true: Iff* $(p, q) \in m(n)$*, the shortest distinguishing word of* p, q *has length* n*. Formally:*

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

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

$n = 0$, **“⇔”.**

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

$n > 0$, **“⇒”.** Then the following holds for some states p, q (see alg. 0, line 5):

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

We will prove: There exists a distinguishing word of length $n - 1$ for p, q, and there is no shorter distinguishing word for p, q.

Looking at eq. 4.1 we observe, 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.

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 precondition no shorter distinguishing word for p, q, making $\sigma w'$ (a) shortest distinguishing word for p, q. $\checkmark$

$n > 0$, **"⇐"**. Then the following holds for some states p, q:

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

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 otherwise 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 3. *If* FindEquivPairs *has done n iterations and terminated (so $\mathfrak{D}(A) = n$), 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-FindEquivPairs(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. $\forall i > n \colon m(i) = \perp$.

This follows directly from while loop and its terminating condition of FindEquivPairs (alg. 0, line 4–7). Given this, we will prove: There exists a shortest distinguishing word of length $n-1$ for some state pair, but a longer such word cannot exist.

Following Lemma 2 and the first observation, 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 $p', q' \in Q$. Following Lemma 2 again, $m(k)$ for some $k > n-1$ would be defined and non-empty, which is contradictory to observations 2 and 3. □

Theorem 3. *Given two DFAs A, A'. If both are accessible and their language is the same ($L(A) = L(A')$), then* FindEquivPairs *runs with the same number of iterations on them ($\mathfrak{D}(A) = \mathfrak{D}(A')$).*

Proof. Starting with the language-equivalence of A and A' we observe, that the start states of both DFAs have the same finishing words.

$$\begin{aligned} & L(A) = L(A') \\ \Rightarrow\ & \{\ w \mid \delta^*(s, w) \in F\ \} = \{\ w \mid \delta'^*(s', w) \in F'\ \} \\ \Rightarrow\ & \forall w \in \Sigma^* \colon \delta^*(s, w) \in F \Leftrightarrow \delta'^*(s', w) \in F' \end{aligned}$$

We extend this to a statement that includes any state visited on the way to F resp. F'. We can see, that those states reached by the same word in A, A' have the same finishing words.

$$\begin{aligned}
&\forall u \in \Sigma^* \colon \exists q, q' \in Q \colon \\
&\qquad \delta^*(s,u) = q \wedge \delta'^*(s',u) = q' \wedge \\
&\qquad (\forall v \in \Sigma^* \colon (\delta^*(q,v) \in F \Leftrightarrow \delta'^*(q',v) \in F')) \\
\Rightarrow\ &\forall u \in \Sigma^* \colon \exists q, q' \in Q \colon \\
&\qquad \delta^*(s,u) = q \wedge \delta'^*(s',u) = q' \wedge \\
&\qquad f(q) = f(q')
\end{aligned}$$

Since we are making a statement about all states reached from s/s' and since all states in A/A' are reachable, we may conclude:

For every state in A/A' there exists a state in the other DFA, such that their finishing words are the same.

$$\begin{aligned}
&\forall q \in Q \colon \exists q' \in Q' \colon f(q) = f(q') \quad \wedge \quad \forall q' \in Q' \colon \exists q \in Q \colon f(q) = f(q') \\
\Rightarrow\ &\{\ f(q) \mid q \in Q\ \} = \{\ f(q') \mid q' \in Q'\ \}
\end{aligned}$$

Since a distinguishing word is defined as being finishing word for one state and for one not (see def. 8), there cannot be a distinguishing word in one of A/A', that is not distinguishing word in the other DFA.

As a consequence both DFAs have the same shortest distinguishing words and thus too the same longest shortest distinguishing word.

If $\mathfrak{D}(A) \neq \mathfrak{D}(A')$ then by Lemma 3 one DFA would have a longer longest shortest distinguishing word, which is not true as proven, thus $\mathfrak{D}(A) = \mathfrak{D}(A')$ must be true. □

Corollary 1. *Our method of creating equivalent state pairs in a DFA does not change the DFAs $\mathfrak{D}$-value.*

Proof. Creating equivalent state pairs does not change the language of a DFA — otherwise the reverse procedure, MinimizeDFA, could not guarantee the language stays the same. So since the language stays the same when creating those pairs, by Theorem 3 the number of FindEquivPairs-iterations will be preserved as well. □

4.2 Adding Unreachable States

Unreachable states are states, which are not reachable from the start state (see def. 1). Consequently there are no restrictions on number and nature of their outgoing transitions. Let us say the added states is q.

Concerning q's ingoing transitions, we only have to ensure, that none of them allows reaching q from the start state. This implicates, that only other unreachable states are qualified as start points of those ingoing transitions.

Note that, given q, σ, we allow only one end point to $\delta(q,\sigma)$. Consequently if we want to add an ingoing transition $\delta(q_u,\sigma) = q$ where q_u is unreachable, then we are stealing the transition from another state $\delta(q_u,\sigma) = q'$. But this is okay, since q' must be an unreachable state too and will stay that way if we remove an ingoing transition.

In this algorithm we assume, that A has no unreachable states.

```
1: function AddUnrStates (A, n_u, c)
2:     U ← ∅                                   ▷ unreachable states in A
3:     for n_u times do
4:         q ← unused state label
5:         Add q to Q
```

```
6:      for q_u, σ in random subset of U × Σ do                ▷ in. transitions
7:          δ(q_u, σ) = q
8:      Σ' ← if c = 1 then Σ else random subset of Σ           ▷ out. transitions
9:      S ← random chosen sample of |Σ'| states from Q
10:     for σ in Σ' do
11:         Remove a state q' from S
12:         δ(q, σ) = q'
13:     Add q to U
14: 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 tasks could be generated automatically. Therefore we discussed prerequisites and requirements to such a program and used 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 randomization and enumeration; 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 method but gave outlook to more efficient variants.

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

When building the task DFA, a question arised concerning the number of iterations by the FindEquivPairs-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 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.

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 DFA drawing.

Our summary on research on DFA generation indicated that efficient - randomized and enumerating - methods to generate DFAs have already been found, where the resulting DFAs were even accessible. An improved version of the associated implementation could make use some of these methods. 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 [9] presented a randomization method that is deemed easy to implement.

Concerning the process of extending solution DFAs to task DFAs, one might ask, how an enumeration algorithm similar to the one for generating solution DFAs could work. In doing so, one might furthermore think about the chance of hitting the same task DFA twice, when the extension algorithm is applied two times on the same DFA.

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.

```
1:  function AreIsomorphic (A_1, A_2)
2:      if |Q_1| ≠ |Q_2| or |F_1| ≠ |F_2| or Σ_1 ≠ Σ_2 then
3:          return false
4:      π(s_1) = s_2                                   ▷ bijection Q_1 → Q_2
5:      O ← ∅                                          ▷ observed states
6:      V ← {s_1}                                      ▷ visited states
7:      q_c ← s_1                                      ▷ current state
8:      while true do
9:          for ((q_1, σ), p_1) in δ_1 do              ▷ iterate through d^+(q_c)
10:             if q_1 ≠ q_c then
11:                 continue
12:
13:             p_2 ← δ_2(π(q_c), σ)
14:             p1marked ← (π(p_1) ≠ ⊥)                ▷ see if p_1, p_2 were "marked" by π
15:             p2marked ← (∃q: π(q) = p2)
16:
17:             if p1marked and p2marked then
18:                 if π(p_1) ≠ p_2 then
19:                     return false
20:             else if ¬p1marked and ¬p2marked then
21:                 π(p_1) = p_2
22:                 if p_1 ∉ V then
23:                     Add p_1 to O
24:             else                                   ▷ one of p_1, p_2 was assigned to some state ≠ p_1 resp. p_2
25:                 return false
26:         if |O| = 0 then
27:             break
28:         Pick and remove q_c from O
29:         Add q_c to V
30:     end
31:     for q_1 in F_1 do
32:         if π(q_1) ∉ F_2 then
33:             return false
34:     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 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 $\delta_2(\pi(q_c), \sigma) = p_2$, 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 one of both states has already been assigned to another state. Thus the construction of a bijection has failed here, since p_1, p_2 should belong together.

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] Jean Berstel, Luc Boasson, Olivier Carton, and Isabelle Fagnot. Minimization of automata, 2010. arXiv:1010.5318.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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`.

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

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

[18] 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`.

[19] William Kocay. The Hopcroft-Tarjan planarity algorithm. October 1993.

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

[21] Frank Lamb. *Industrial Automation: Hands On*. McGraw-Hill Education, 2013. URL: `https://books.google.de/books?id=H977GaxHwaAC`.

[22] Wim Martens and Thomas Schwentick. Theoretische Informatik I SS18. Lecture notes, 2018.

[23] Cyril Nicaud. *Étude du Comportement en Moyenne des Automates Finis et des Langages Rationnels*. PhD thesis, Université Paris 7, 2000.

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

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

[26] 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.

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

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

Erklärung

Hiermit versichere ich, Gregor Hans Christian Sönnichsen, dass ich die vorliegende Arbeit selbstständig 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.

Stedesand, den 20. Mai 2020.

Gregor Hans Christian Sönnichsen