

# Generation of DFA Minimization Problems

Gregor H. C. Sönnichsen

February 5, 2020

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Preliminaries . . . . .	2
1.1.1	Deterministic Finite Automatons . . . . .	2
1.1.2	Minimal DFAs . . . . .	3
1.1.3	Isomorphy of DFAs . . . . .	4
1.1.4	Duplicate states . . . . .	4
1.1.5	The minimization algorithm . . . . .	4
1.2	Requirements analysis . . . . .	6
1.2.1	Example of a DFA minimization task for students . . . . .	6
1.2.2	Definition and evaluation of possible requirements . . . . .	6
1.3	Approach and general algorithm . . . . .	8
<b>2</b>	<b>Building solution DFAs</b>	<b>10</b>
2.1	Using trial and error . . . . .	10
2.1.1	Ensuring $A_{test}$ is minimal and $\mathcal{D}(A_{test})$ is correct . . . . .	11
2.1.2	Ensuring $A_{test}$ is planar . . . . .	12
2.1.3	Ensuring $A_{test}$ is unused . . . . .	12
2.1.4	Option 1: Generating $A_{test}$ via Randomness . . . . .	12
2.1.5	Option 2: Generating $A_{test}$ via Enumeration . . . . .	13
2.1.6	Ideas for more efficiency . . . . .	16
2.2	Building directly minimal DFAs . . . . .	16
2.2.1	Research . . . . .	16
2.2.2	Building $m(i)$ bottom up . . . . .	16
<b>3</b>	<b>Extending solution DFAs to task DFAs</b>	<b>17</b>
3.1	Adding duplicate states . . . . .	17
3.1.1	Adding duplicate states does not change L . . . . .	19
3.1.2	Adding duplicate states does not change $\mathcal{D}$ . . . . .	19
3.2	Adding unreachable states . . . . .	21
<b>4</b>	<b>Notes on the implementation</b>	<b>23</b>
<b>5</b>	<b>Conclusion</b>	<b>24</b>

# Chapter 1

## Introduction

- study computer science
- theoretical informatics
- automata theory
- value of this theory
- typical topics, why typical
- why automation

This work lays out the theory for a program solving this task. As a consequence, parameters, which are sensible as user input, will be incorporated in problem definitions. In addition, when evaluating possible algorithms, we will take their usability in a practical use case into account. Furthermore additional theory will be discussed, to enhance usability.

### 1.1 Preliminaries

We start with defining preliminary theoretical foundations.

#### 1.1.1 Deterministic Finite Automatons

A 5-tuple  $A = (Q, \Sigma, \delta, s, F)$  with  $Q$  being a finite set of *states*,  $\Sigma$  a finite set of *alphabet symbols*,  $\delta: Q \times \Sigma \rightarrow Q$  a *transition function*,  $s \in Q$  a *start state* and  $F \subseteq Q$  *final states* is called *deterministic finite automaton* (DFA) [3, 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  $\sigma$ . We define the *extended transition function*  $\delta^*: Q \times \Sigma^* \rightarrow Q$  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 that DFA is defined as  $L(A) = \{ w \mid \delta^*(s, w) \in F \}$  [3, pp. 49-50. 52].

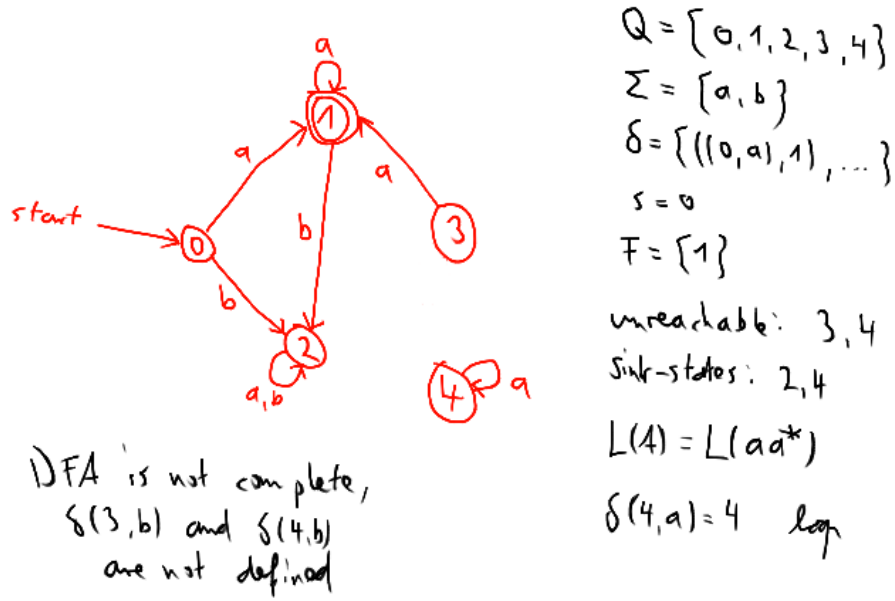


Figure 1.1: An example DFA and its properties.

Given a state  $q \in Q$ . With  $d^-(q)$  we denote the set of all *incoming* transitions  $\delta(q', \sigma) = q$  of  $q$ . With  $d^+(q)$  we denote the set of all *outgoing* transitions  $\delta(q, \sigma) = q'$  of  $q$  [1, pp. 2-3]. If a transition is of the form  $\delta(q, \sigma) = q$ , then we say that  $q$  has a *loop*.

**Definition 1.** 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 say  $A$  is *accessible* [1, p. 2].

A DFA is called *complete* iff for all states, every symbol of the alphabet is used on an outgoing transition:  $\forall q \in Q: \forall \sigma \in \Sigma: \exists p \in Q: \delta(q, \sigma) = p$ . Note, that every incomplete DFA can be converted to a complete one by adding a so called *dead state* [3, p. 67]. The resulting automaton has the same language.

### 1.1.2 Minimal DFAs

This section closely follows [5, pp. 42-45]. 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.

The *Nerode-relation*  $\equiv_L \subseteq \Sigma^* \times \Sigma^*$  of a language  $L$  with alphabet  $\Sigma$  is defined as follows:

$$x \equiv_L y \Leftrightarrow_{def} \forall z \in \Sigma^*: (xz \in L \Leftrightarrow yz \in L)$$

The Nerode-relation of a DFA  $A$  is the the Nerode-relation of its language:  $\equiv_{L(A)}$ . If the context makes it clear, than we will shorten the notation of a equivalence class  $[x]_{\equiv_L}$  with  $[x]$ .

The *equivalence class automaton*  $A_L = (Q_L, \Sigma_L, \delta_L, s_L, F_L)$  to a regular language  $L$  with alphabet  $\Sigma$  is defined as follows:

- $Q_L = \{ [x] \mid x \in \Sigma^* \}$
- $\Sigma_L = \Sigma$
- $\delta_L([x], \sigma) = [x\sigma], \forall x \in \Sigma^*, \forall \sigma \in \Sigma$
- $s = [\varepsilon]$
- $F = \{ [x] \mid x \in L \}$

**Theorem 1.** *Given a language  $L$ , then the equivalence class automaton  $A_L$  is minimal.*

### 1.1.3 Isomorphism 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* ( $A_1 \cong A_2$ ), iff:

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

**Theorem 2.** [5, p. 45] *Every minimal DFA is unique except for isomorphism.*

**Corollary 1.** *Every minimal DFA  $A$  is isomorph to its corresponding equivalence class automaton  $A_{L(A)}$ . **Gregor:** All min. DFAs are ism. to each other, including  $A_L$*

**Gregor:** Write down isomorphism test. Maybe discuss faster methods here? Look for faster methods in general?

### 1.1.4 Duplicate states

**Definition 2** (Duplicate States). [3, p. 154] Two states  $q_1, q_2 \in Q$  of a DFA  $A = (Q, \Sigma, \delta, s, F)$  are called *duplicates* of each other, iff  $d_A(q_1, q_2)$  is true, whereas

$$q_1 d_A q_2 \iff_{def} \forall z \in \Sigma^*: (\delta^*(q_1, z) \in F \iff \delta^*(q_2, z) \in F)$$

Note that the relation  $d_A$  is indeed an equivalence relation.

### 1.1.5 The minimization algorithm

This minimization algorithm requires a complete DFA and works in four major steps, removing essentially states in such a way, that no unreachable and no duplicate states are left.

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

1: **function** COMPUTEUNREACHABLESTATES( $A$ )

```

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:      $N \leftarrow \{ p \mid \exists q \in O \ \sigma \in \Sigma: \ \delta(q, \sigma) = p \ \wedge \ p \notin O \cup D \}$ 
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 REMOVEUNREACHABLESTATES( $A, U$ )
2:   for  $q$  in  $U$  do
3:     if  $q \in F$  then
4:        $F \leftarrow F \setminus \{q\}$ 
5:        $\delta \leftarrow \delta \setminus \{ ((q_1, \sigma), q_2) \in \delta \mid q_1 = q \vee q_2 = q \}$ 
6:   return  $A$ 

```

3. Compute all non-duplicate states ( $\neg d_A(p, q)$ ) via the MINIMIZATIONMARK-algorithm.

```

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

```

4. Merge all duplicate state pairs, which are exactly those, that are not in  $\neg d_A$ .

```

1: function MINIMIZATIONMERGE( $A, \neg d_A$ )
2:   compute  $d_A$ 
3:   while  $d_A \neq \emptyset$  do
4:      $(p, q) \in d_A$ 
5:      $d_A \leftarrow d_A \setminus \{(p, q)\}$ 
6:     if  $p \neq q$  then
7:       exchange all occurrences of  $q$  in  $A$  and  $d_A$  by  $p$ 
8:   return  $A$ 

```

**Theorem 3.** *The minimization algorithm computes a minimal DFA to its input DFA.*

The definition of this DFA minimization algorithm is inspired by Schönig [5, p. 46].

When looking at MINIMIZATIONMARK, 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  $m: \mathbb{N} \rightarrow \mathcal{P}(Q \times Q)$ :

```

1: function  $m$ -MINIMIZATIONMARK( $(Q, \Sigma, \delta, s, F)$ )
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:      $m(i) \leftarrow \{(p, q) \mid (p, q) \notin \bigcup m(\cdot) \wedge \exists \sigma \in \Sigma: (\delta(p, \sigma), \delta(q, \sigma)) \in m(i-1)\}$ 
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 in exchange.

We will denote the number of iterations done by MINIMIZATIONMARK on an DFA  $A$  as  $\mathcal{D}(A)$ . Note that  $\mathcal{D}(A) = \max n \in \mathbb{N} \mid m(n) \neq \emptyset$ . **Gregor:** Does that note maybe fit very well to the proof of lemma 2?

## 1.2 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.

### 1.2.1 Example of a DFA minimization task for students

Figures 1.2 and 1.3 show such a task and solution. We will call the minimal automaton *solution DFA* ( $A_{sol}$ ) and the DFA given in the task *task DFA* ( $A_{task}$ ).

The table  $T$  displayed in the solution is nothing else but a visualization of the function  $m$ , whereas  $T(q_0, q_1) = i \Leftrightarrow (q_0, q_1) \in m(i)$ . The solution corresponds with the formal definition of the minimization algorithm, but focuses rather on the concepts than demanding formal breakdown.

- present a typical task and its solution (text, image, table)
- search for typical test in text standard work books
- name its elements
- clarify two parts: solution, task
- clarify four parts from task to solution: find unreachable, delete them, find duplicates, merge them
- what are the difficulties of this tasks?

### 1.2.2 Definition and evaluation of possible requirements

Dismissed:

- $h(A_{sol}, A_{task}) = |Q_{task}| / |Q_{sol}|$
- number of transitions

Given the following DFA:

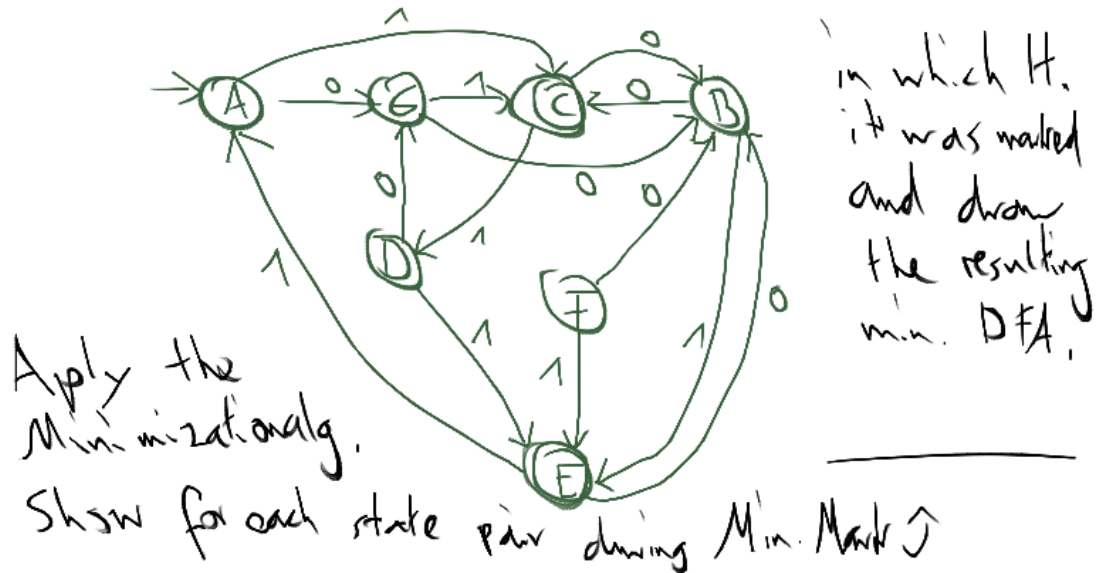


Figure 1.2: An example DFA minimization task.

Step 1: Delete unreachable states.  
F is unreachable.

Step 2: Min Mark. & merge dupl. states

	A	B	C	D	E	G
A	///	1	0		0	2
B	///	///	0	1	0	1
C	///	///	///	0		0
D	///	///	///	///	0	2
E	///	///	///	///	///	0
G	///	///	///	///	///	///

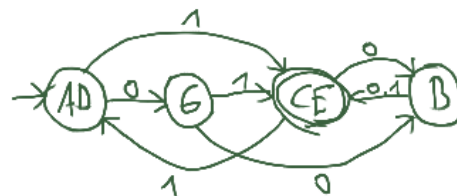


Figure 1.3: Solution to the DFA minimization task in fig. 1.2.



- max degree of a node (Why not this?)
- Does GraphViz have a heuristic?

Accepted solution DFA criteria:

- > minimal
- > number of states
- > number of MINIMIZATIONMARK iterations ( $\mathcal{D}(A_{sol})$ )
- > alphabet size
- > number of accepting states
- > planarity (can be checked in  $O(|Q_{sol}|)$ )
- > completeness (for easier further processing)
- >  $A_{sol}$  is unused (regarding all previously generated solution DFAs)

**Definition 3** (Unused DFAs). A DFA  $A$  is *unused* regarding a set of *used* DFAs  $U$ , if  $A$  is not isomorph to any DFA in  $U$ .

Accepted task DFA criteria:

- >  $L(A_{sol}) = L(A_{task})$
- >  $\mathcal{D}(A_{sol}) = \mathcal{D}(A_{task})$
- > number of duplicate states
- > number of unreachable states
- > alphabet size
- > planarity (can be checked in  $O(|Q_{task}|)$ )
- > completeness (for MINIMIZATIONMARK-algorithm to work)

### 1.3 Approach and general algorithm

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

It follows that  $\mathcal{D}$  and  $L$  of both DFAs will be set when building  $A_{sol}$ . As a consequence we need to ensure that adding duplicate and unreachable state does neither change  $\mathcal{D}(A_{task})$  nor  $L(A_{task})$  in comparison to  $A_{sol}$ . We will do this during the discussion of step 2.

Here follow problem definitions for the two steps, which specify all needed informations. **Gregor:** [Hidden formulation here](#)

**Definition 4** (BuildNewMinimalDFA).

**Given:**

$$\begin{aligned} q, a, f, m_{min}, m_{max} &\in \mathbb{N}, \\ p &\in \{0, 1\} \end{aligned}$$

**Request:**

Let  $A_{sol} = (Q, \Sigma, \delta, s, F)$  be a DFA, such that

$$|Q| = q, |\Sigma| = a, |F| = f,$$

$$m_{min} \leq \mathcal{D}(A_{sol}) \leq m_{max},$$

$A_{sol}$  is planar iff  $p = 1$  and

the language of  $L(A)$  is unequal to any DFA used before.

Return  $A_{sol}$ , if it exists,  $\perp$  otherwise.

**Definition 5** (ExtendMinimalDFA).

**Given:**

$$\begin{aligned} A_{sol} &= (Q, \Sigma, \delta, s, F) \in \mathcal{A}_{min}, \\ p &\in \{0, 1\}, \\ d, u &\in \mathbb{N} \end{aligned}$$

**Request:**

A DFA  $A_{task} = (Q', \Sigma', \delta', s', F')$  with reachable duplicate states  $q_1 \dots q_d$  and unreachable states  $p_1 \dots p_u$ , such that

$$Q = Q' \cup \{q_1, \dots q_d, p_1 \dots p_u\},$$

$$\Sigma = \Sigma', s = s',$$

$$F \subseteq F',$$

$A_{task}$  is planar iff  $p = 1$ ,

$$L(A_{sol}) = L(A_{task}) \text{ and } \mathcal{D}(A_{sol}) = \mathcal{D}(A_{task}).$$

The main algorithm will then simply be:

- 1: **function** GENERATEDFAMINIMIZATIONPROBLEM( $q, a, f, m_{min}, m_{max}, p_1, p_2, d, u$ )
- 2:      $A_{sol} \leftarrow \text{BUILDNEWMINIMALDFA}(q, a, f, m_{min}, m_{max}, p_1)$
- 3:      $A_{task} \leftarrow \text{EXTENDMINIMALDFA}(A_{sol}, p_2, d, u)$
- 4:     **return**  $A_{sol}, A_{task}$

# Chapter 2

## Building solution DFAs

We want an algorithm for DFA generation that fulfills the following conditions (see 1.2.2):

- > minimal
- > number of states
- > number of MINIMIZATIONMARK iterations ( $\mathcal{D}(A_{sol})$ )
- > alphabet size
- > number of accepting states
- > planarity (can be checked in  $O(|Q_{sol}|)$ )
- > completeness (for easier further processing)
- >  $A_{sol}$  is unused (regarding all previously generated solution DFAs)

These conditions have been formally subsumed as BuildNewMinimalDFA-problem (see def. 4). We consider different approaches to solve this problem, of which those using trial-and-error will be discussed most broadly.

### 2.1 Using trial and error

We will develop an algorithm that makes partly use of the trial-and-error paradigm to find matching DFAs. The approach here is 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 unused. If this is the case,  $A_{test}$  will be returned, if not, new test DFAs are generated until all tests pass.

By constructing test DFAs with already correct alphabet size and number of (final) states we are able to subdivide the search space of DFAs in advance into much smaller pieces.

**Gregor:** How much smaller?

1: **function** BUILDNEWMINIMALDFA-1 ( $q, a, f, m_{min}, m_{max} \in \mathbb{N}, p \in \{0, 1\}$ )

```

2:   while True do
3:       generate DFA  $A_{test}$  with  $|Q|, |\Sigma|, |F|$  matching  $q, a, f$ 
4:       if  $A_{test}$  not minimal or not  $m_{min} \leq \mathcal{D}(A_{test}) \leq m_{max}$  then
5:           continue
6:       if  $p = 1$  and  $A_{test}$  is not planar then
7:           continue
8:       if  $A_{test}$  was used before 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 automaton with given restrictions of  $|Q|, |\Sigma|$  and  $|F|$ .

### 2.1.1 Ensuring $A_{test}$ is minimal and $\mathcal{D}(A_{test})$ 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 isomorphism test. However it is sufficient to ensure, that no duplicate or unreachable states exist. **Gregor:** minimality planarity complete under isomorphism

To get  $\mathcal{D}(A_{test})$ , we have to run MINIMIZATIONMARK entirely anyway. Hence we can combine the test for duplicate states with computing the DFA's  $\mathcal{D}$ -value:

```

1: function HASDUPLICATESTATES( $A$ )
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:        $M' \leftarrow \{(p, q) \mid (p, q) \notin M \wedge \exists \sigma \in \Sigma: (\delta(p, \sigma), \delta(q, \sigma)) \in M\}$ 
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 MINIMIZATIONMARK computes all non-duplicate state pairs  $\neg d_A$ , we test in line 9, whether there is a pair of distinct states not in  $\neg d_A$ .

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

```

1: function HASUNREACHABLESTATES( $A$ )
2:   return  $|\text{COMPUTEUNREACHABLESTATES}(A)| > 0$ 

```

**Gregor:** Is there a more efficient method? Since we actually need to know of only one unreachable state.

### 2.1.2 Ensuring $A_{test}$ is planar

There exist several algorithms for planarity testing of graphs. In this work, the library *pygraph*<sup>1</sup> has been used, which implements the Hopcroft-Tarjan planarity algorithm. More information on this can be found for example in this [4] introduction from William Kocay. The original paper describing the algorithm is [2].

### 2.1.3 Ensuring $A_{test}$ is unused

In our requirements we stated, that we wanted the generated solution DFA to be unused with regards to all previous generated solution DFAs. This implies the need of a database, that allows saving single DFAs and loading DFAs. We name this database *DB1*. Assuming the database is relational, the following scheme is proposed:

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

With this scheme we can fetch once all DFAs matching the search parameters. Thus we need not fetch all used 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:

```

1: function BUILDNEWMINIMALDFA-2 ( $q, a, f, m_{min}, m_{max}, p$ )
2:    $l \leftarrow$  all DFAs in DB1 matching  $q, a, f, m_{min}, m_{max}, p$ 
3:   while True do
4:     ...
5:     if  $A_{test}$  is isomorph to any DFA in  $l$  then
6:       continue
7:     save  $A_{test}$  and its respective properties in DB1
8:     return  $A_{test}$ 

```

### 2.1.4 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.

Corollary 1 tells us, that the states names are irrelevant for the minimality of a DFA, therefore we will give our generated DFAs simply the states  $q_0, \dots, q_{q-1}$ . For alphabet symbols this is not given. But since we **Gregor: TODO minimality and planarity complete under isomorphy**

We can state, that our start state is  $q_0 \in Q$ , since we apply an isomorphism to every that, such that its start state is relabeled to  $q_0$ .

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

---

<sup>1</sup><https://github.com/jciskey/pygraph>

The transition function has to make the DFA complete, so we have to choose an “end” state for every combination in  $Q \times \Sigma$ . There is no restriction as to what this end state shall be, so given  $q \in Q$  and  $\sigma \in \Sigma$  we can randomly choose an end state from  $Q$ .

With defining how to compute  $\delta$  we have covered all elements of a DFA.

```

1: function BUILDNEWMINIMALDFA-3A ( $q, a, f, m_{min}, m_{max}, p$ )
2:    $l \leftarrow$  all DFAs in DB1 matching  $q, a, f, m_{min}, m_{max}, p$ 
3:    $Q \leftarrow \{q_0, \dots, q_{q-1}\}$ 
4:    $\Sigma \leftarrow \{\sigma_0, \dots, \sigma_{a-1}\}$ 
5:   while True do
6:      $\delta \leftarrow \emptyset$ 
7:     for  $q$  in  $Q$  do
8:       for  $\sigma$  in  $\Sigma$  do
9:          $q' \leftarrow$  random chosen state from  $Q$ 
10:         $\delta \leftarrow \delta \cup \{(q, \sigma), q'\}$ 
11:     $s \leftarrow 0$ 
12:     $F \leftarrow$  random sample of  $f$  states from  $Q$ 
13:     $A_{test} \leftarrow (Q, \Sigma, \delta, s, F)$ 
14:    if  $A_{test}$  not minimal or not  $m_{min} \leq \mathcal{D}(A_{test}) \leq m_{max}$  then
15:      continue
16:    if  $p = 1$  and  $A_{test}$  is not planar then
17:      continue
18:    if  $A_{test}$  is isomorph to any DFA in  $l$  then
19:      continue
20:    save  $A_{test}$  and its respective properties in DB1
21:    return  $A_{test}$ 

```

### 2.1.5 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  $\delta$ , we could just enumerate through all possible final state sets and transition functions.

Both enumerations are finite, given  $q$  and  $a$ . Having a requirement of  $f$  final states, then  $q$  choose  $f$  is the number of possible  $F$ -configurations. On the other hand there are  $q^{qa}$  possible  $\delta$ -configurations. **Gregor:** why

We will represent the state of an enumeration with two bit-fields  $b_f$  and  $b_t$ . The first bit-field shall have  $q$  Bits, whereas Bit  $b_f[i] \in \{0, 1\}$  represents the information, whether  $q_i$  is a final state or not. The second bit-field shall have  $q * a * \log_2(q)$  Bits, such that Bit  $b_t[i * a + j] = k$  says, that  $\delta(q_i, \sigma_j) = q_k$ . These semantics are illustrated in figure 2.1.

Given an enumeration state  $b_f, b_t$  and  $q, a, f$  we will then compute the next DFA based on this state as follows. We will treat both bit-fields as numbers,  $b_f$  as binary and  $b_t$  as  $\log_2(q)$ -ary. To get to the next DFA, we will first increment  $b_t$  by 1. If  $b_t = 1 \dots 1$ , then we increment  $b_f$  until it contains  $f$  ones (again) and set  $b_t$  to  $0 \dots 0$ . This behaviour is summarized in the following algorithm: **Gregor:** Clarify what

Given:  $q=4, a=2, f=3$

example  $b_f$ :

1101

so

$$b_f[0] = 1$$

$$b_f[1] = 1$$

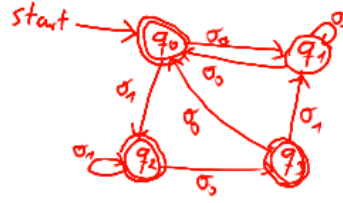
$$b_f[2] = 0$$

$$b_f[3] = 1$$

$$F = \{q_0, q_2, q_3\}$$

example  $b_t$ :

$$\begin{array}{c|c|c|c} q_0 & q_1 & q_2 & q_3 \\ \hline \sigma_0 \sigma_1 & \sigma_0 \sigma_1 & \sigma_0 \sigma_1 & \sigma_0 \sigma_1 \\ \hline q_1 q_2 & q_0 q_1 & q_3 q_2 & q_0 q_1 \\ \hline 01 & 10 & 00 & 01 & 11 & 10 & 00 & 01 \end{array}$$



so e.g.

$$\delta(q_0, \sigma_0) = q_1$$

$$\delta(q_2, \sigma_1) = q_2$$

...

Figure 2.1: Example for two possible configurations of the bit-fields  $b_f$  and  $b_t$  given  $q, a$  and  $f$ . Below the corresponding DFA is drawn.

happens at 11111...

```

1: function INCREMENTENUMPROGRESS ( $b_f, b_t, q, a, f$ )
2:   add 1 to  $(b_t)_2$ 
3:   if  $b_t = 0 \dots 0$  then
4:     while  $\#_1(b_f) \neq f$  do
5:       add 1 to  $(b_f)_2$ 
6:       if  $b_f = 0 \dots 0$  then
7:         return  $\perp$ 
8:        $b_t = 0 \dots 0$ 
9:   return  $b_f, b_t$ 

```

The example in figure 2.2 illustrates such increments.

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

```

1: function DFAFROMENUMPROGRESS ( $b_f, b_t, f$ )
2:    $Q \leftarrow \{q_0, \dots, q_{q-1}\}$ 
3:    $\Sigma \leftarrow \{\sigma_0, \dots, \sigma_{a-1}\}$ 
4:    $\delta \leftarrow \emptyset$ 
5:   for  $i$  in  $[0, \dots, q-1]$  do
6:     for  $j$  in  $[0, \dots, a-1]$  do
7:        $\delta \leftarrow \delta \cup \{(q_i, \sigma_j), q_{b_t[i*a+j]}\}$ 
8:    $s \leftarrow q_0$ 
9:    $F \leftarrow \{q_i | i \in [0, \dots, q-1] \wedge b_f[i] = 1\}$ 
10:  return  $(Q, \Sigma, \delta, s, F)$ 

```

Given:  $q=4, a=2, f=3$

example  $b_t$ :

example  $b_f$ :

1101

$$\begin{array}{cccc}
 q_0 & q_1 & q_2 & q_3 \\
 \sigma_0 & \sigma_1 & \sigma_0 & \sigma_1 \\
 q_1 & q_2 & q_0 & q_1 \\
 q_3 & q_2 & q_0 & q_1 \\
 01 & 10 & 00 & 01 \\
 11 & 10 & 00 & 01 \\
 + & & & 1 \\
 = & 01 & 10 & 00 \\
 & 01 & 11 & 10 \\
 & 00 & 10 & \\
 & q_1 & q_2 & q_0 \\
 & q_1 & q_3 & q_2 \\
 & q_0 & q_1 & q_2
 \end{array}$$

Assuming  $b_t = 11\ 11\ 11\ 11\ 11\ 11\ 11\ 11$  before incr.  
then  $b_f$  is incr. until it has  $f$  1's again

1101  $\rightarrow$  1110

and  $b_t$  is set to 00 00 00 00 00 00 00 00

Figure 2.2: The upper half shows how a  $b_t$ -increment results in a change in the resulting DFAs transition function:  $\delta(q_3, \sigma_1) = q_1$  becomes  $\delta(q_3, \sigma_1) = q_2$ . The lower half shows what happens, if  $b_t$  has reached its end.

The initial bit-field values are each time  $0 \dots 0$ . Note how construction and use of these bit-fields results in DFAs with correct alphabet size and number of (final) states. We define  $Q$  and  $\Sigma$  as in the random generation method. An enumeration can finish either because a matching DFA has been found or all DFAs have been enumerated **Gregor: More, beautiful, explanation. Find proper place.**

Once the enumeration within a call of BUILDNEWMINIMALDFA has been finished, it is reasonable to save the progress (meaning the current content of  $b_f, b_t$ ), 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, whereas all so far found DFAs would be found used. Thus we introduce a second database  $DB2$  with the following table:

$ Q_A $	$ \Sigma_A $	$b_f$	$b_t$
---------	--------------	-------	-------

We reduce the enumeration room for each calculation.

- 1: **function** BUILDNEWMINIMALDFA-3B ( $q, a, f, m_{min}, m_{max}, p$ )
- 2:  $l \leftarrow$  all DFAs in DB1 matching  $q, a, f, m_{min}, m_{max}, p$
- 3:  $b_f, b_t \leftarrow$  load enumeration progress for  $q, a, f, p$  from DB2
- 4: **while** True **do**
- 5:     **if**  $b_f, b_t$  is finished **then**
- 6:         save  $b_f, b_t$
- 7:         **return**  $\perp$
- 8:      $A_{test} \leftarrow$  next DFA based on  $b_f, b_t$



```

9:      if  $A_{test}$  not minimal or not  $m_{min} \leq \mathcal{D}(A_{test}) \leq m_{max}$  then
10:         continue
11:      if  $p = 1$  and  $A_{test}$  is not planar then
12:         continue
13:      if  $A_{test}$  is isomorph to any DFA in  $l$  then
14:         continue
15:      save  $b_f, b_t$  in DB2
16:      save  $A_{test}$  and its respective properties in DB1
17:      return  $A_{test}$ 

```

### 2.1.6 Ideas for more efficiency

incrementing final state binary faster in enum-alternative  
 speed up isomorphy test  
 rewrite everything in C  
 solve P vs NP

## 2.2 Building directly minimal DFAs

### 2.2.1 Research

### 2.2.2 Building $m(i)$ bottom up

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

# Chapter 3

## Extending solution DFAs to task DFAs

Given a solution DFA  $A_{sol}$  we have determined the following requirements for generating a task DFA  $A_{task}$  in our requirements analysis (see 1.2.2):

- >  $L(A_{sol}) = L(A_{task})$
- >  $\mathcal{D}(A_{sol}) = \mathcal{D}(A_{task})$
- > number of duplicate states
- > number of unreachable states
- > alphabet size
- > planarity (can be checked in  $O(|Q_{task}|)$ )
- > completeness (for MINIMIZATIONMARK-algorithm to work)

In order to fulfill these requirements when adding new elements to the given minimal automaton  $A_{sol}$ , we simply look at how duplicate and unreachable states are removed by the minimization algorithm, such that we can deduce from their properties, which restrictions are given for adding such elements. We will show for both classes of addable elements, that they do not change the DFAs language and its  $\mathcal{D}$ -value.

**Gregor:** Adding unreachable states is essentially just talking about that special equivalence class. Think and tell more about this

### 3.1 Adding duplicate states

Firstly, let us state that since unreachable states are removed first in the minimization algorithm, we may assume that every state, that is duplicated, is reachable.

**Gregor:** hidden definition: correct duplication

Step 3 and 4 of the minimization algorithm are concerned with detection and elimination of duplicate states. How do we add duplicate states to a DFA?

Consider the properties a duplicate state, say  $q_d$ , must have. It is in particular duplicate to *another* state, we call it  $q_o$ . We call the new, by  $q_d$  extended DFA,  $A$ .

**Outgoing transitions** We know that  $q_d, q_o$  are duplicates, iff  $\forall \sigma \in \Sigma: [\delta(q_d, \sigma)]_{d_A} = [\delta(q_o, \sigma)]_{d_A}$ . Thus, when adding some  $q_d$ , we have to choose for each symbol  $\sigma \in \Sigma$  at least one transition from the following set:

$$P_\sigma = \{ ((q_d, \sigma), p) \mid p \in [\delta(q_o, \sigma)]_{d_A} \}$$

Since the solution DFA is complete, we know that every  $P_\sigma \neq \emptyset$ .

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

**Ingoing transitions** The ingoing transitions of  $q_d$  are not directly restricted through the duplicateness of  $q_d$  and  $q_o$ .

First of all, we know, that  $q_o$  is reachable. We then need to give  $q_d$  at least one ingoing transition. Doing this, we have to ensure, that any state  $s$ , that gets such an outgoing transition to  $q_d$  remains in its solution equivalence class.

Thus a fitting state  $s$  has to have a transition to some state in  $[q_d]_{d_A} = [q_o]_{d_A}$  already. So, given a state  $s$  with  $((s, \sigma), t)$  and  $t \in [q_o]_{d_A}$ , we can add  $((s, \sigma), q_d)$ .

But this would make our new DFA a NFA. As a consequence we have to remove the original transition  $((s, \sigma), t)$  each time we add an ingoing transition for a newly created duplicate state.

So we have to choose at least one transition of

$$\{ ((s, \sigma), q_d) \mid \delta(s, \sigma) \in [q_o]_{d_A} \}$$

If a  $((s, \sigma), q_d)$  is chosen, remove  $((s, \sigma), t)$ . This leads us to the requirement, that the equivalence class of any  $q_o$  has to contain at least one state with at least 2 ingoing transitions (see fig. 3.1). We establish the following notion to pin down this restriction:

$$\text{duplicatable}(q_o) = 1 \Leftrightarrow_{\text{def}} (\exists q \in [q_o]_{d_A} : |d^-(q)| \geq 2)$$

**Gregor:** Talk somewhere about eq. automaton and extending it. An eq. class of reach. q's can be max.  $|\Sigma|$  big. From this can compute the max. number of dupl. states which can be added.

```

1: function ADDDUPLICATESTATES ( $A_{sol}, d$ )
2:    $eq\_classes \leftarrow \{ \{q\} \mid q \in Q \}$ 
3:    $eq\_class(q) = C$  such that  $C \in eq\_classes$  and  $q \in C$ 
4:    $\#_{d^-}(q) = |d^-(q)|$ 
5:
6:   for  $d$  times do
7:
8:      $q_o \leftarrow \perp$ 
9:     for  $q$  in  $C$  do
10:       if  $\#_{d^-}(q) \geq 2$  then
11:          $q_0 \leftarrow$  random chosen state from  $eq\_class(q)$ 
12:         break
13:       if  $q_o = \perp$  then
14:         return  $\perp$ 
15:
16:    $q_d \leftarrow \max Q + 1$ 
17:    $Q \leftarrow Q \cup \{q_d\}$ 

```

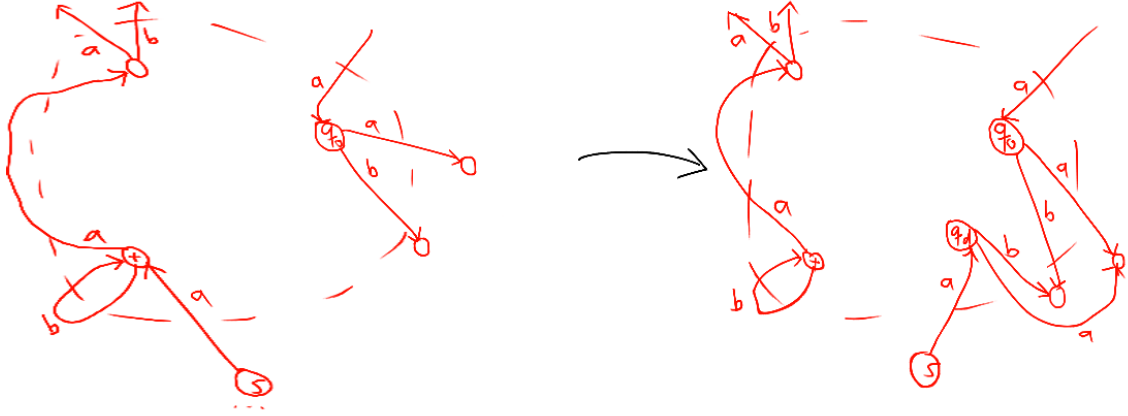


Figure 3.1: If an equivalence class (here denoted by the states in the dashed area) contains a state with 2 or more incoming transitions (in this case  $t$ ), then a state duplicate to any of class states may be added. Here  $q_d$  is duplicate to  $q_o$  and is “stealing” the incoming transition  $\delta(s, a)$  from  $t$ .

```

18:
19:   for  $\sigma$  in  $\Sigma$  do
20:      $\delta(q_d, \sigma) = \text{random chosen state from } eq\_class(\delta(q_o, \sigma))$ 
21:
22:    $O \leftarrow \{ ((s, \sigma), t) \in \delta \mid t \in eq\_class(q_o) \wedge \#_{d^-}(t) \geq 2 \}$ 
23:    $C \leftarrow \text{random sample of at least one transition from } O$ 
24:   for  $((s, \sigma), t)$  in  $C$  do
25:      $\delta \leftarrow \delta \setminus \{((s, \sigma), t)\}$ 
26:      $\delta \leftarrow \delta \cup \{((s, \sigma), q_d)\}$ 
27:      $\#_{d^-}(t) \leftarrow \#_{d^-}(t) - 1$ 
28:      $\#_{d^-}(q_d) \leftarrow \#_{d^-}(q_d) + 1$ 
29:   return  $A$ 

```

### 3.1.1 Adding duplicate states does not change $L$

p. 159 Hopcroft

### 3.1.2 Adding duplicate states does not change $\mathcal{D}$

To prove this statement, we will prove two minor propositions first.

**Lemma 1** (Semantics of  $(p, q) \in m(n)$ ).

$$(p, q) \in m(n) \iff \exists w \in \Sigma^*: |w| = n \wedge$$

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

*Proof.* See TI-Lecture ch. 4 “Minimization” p. 18. □

**Lemma 2** (Semantics of  $\mathcal{D}(A) = n$ ).

$$\mathcal{D}(A) = n \Rightarrow$$

$$n = \max_{n \in \mathbb{N}} \exists p, q \in Q \exists w \in \Sigma^*: |w| = n - 1 \wedge (\delta^*(p, w) \in F \Leftrightarrow \delta^*(q, w) \notin F)$$

*Proof.* Via direct proof.

Assume  $m\text{-MINIMIZATIONMARK}(A)$  has done  $n$  iterations (so  $\mathcal{D}(A) = n$ ). We then know, that

- $\forall i \in [0, n - 1]: m(i) \neq \emptyset$
- $m(n) = \emptyset$

$m\text{-MINIMIZATIONMARK}(A)$  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. The terminating condition is  $m(i) \neq \emptyset$ ; thus follows the second point.

Recall the statement from lemma 1:

$$(p, q) \in m(n) \iff \exists w \in \Sigma^*: |w| = n \wedge (\delta^*(p, w) \in F \Leftrightarrow \delta^*(q, w) \notin F)$$

Following this lemma and having  $m(n - 1) \neq \emptyset$  in mind, we can deduce that there exists at least one word  $w \in \Sigma^*$  with  $|w| = n - 1$  such that for two  $p, q \in Q: (\delta^*(p, w) \in F \wedge \delta^*(q, w) \notin F)$ .

There cannot be any two states  $p', q' \in Q$  and a word  $w' \in \Sigma^*$  with  $|w'| > n - 1$  fulfilling this property. We could write  $w'$  as  $u'v'$  with  $|v'| = n$ . Then  $m(n)$  would be non-empty, which is contradictory. □

**Theorem 4.** *Adding duplicate states to an automaton  $A$  does not increase the number of iterations in the MINIMIZATIONMARK-algorithm for  $A$ .*

*Proof.* Proof per contradiction.

Let's assume adding duplicate states  $q_d^1, \dots, q_d^n$  to a given automaton  $A = (Q, \Sigma, \delta, s, F)$  results in an automaton  $A' = (Q', \Sigma, \delta', s, F')$  whereas  $\mathcal{D}(A) < \mathcal{D}(A')$ .

Concerning  $A'$  we can say the following:

- $Q' = Q \cup \{q_d^1, \dots, q_d^n\}$
- W.l.o.g.  $\exists q_o^1 \in Q: \exists q_o^2 \dots q_o^n \in Q: d'_A(q_o^1, q_d^1), \dots, d'_A(q_o^n, q_d^n)$

Let us furthermore say that  $\mathcal{D}(A) = i$  and  $\mathcal{D}(A') = j$ . Recall now lemma 2:

$$\mathcal{D}(A) = n \Rightarrow n = \max_{n \in \mathbb{N}} \exists p, q \in Q \exists w \in \Sigma^*: |w| = n - 1 \wedge (\delta^*(p, w) \in F \Leftrightarrow \delta^*(q, w) \notin F)$$

According to this lemma there must be a pair  $s, t \in Q'$  to which exists a word  $w \in \Sigma^*$ ,  $|w| = j - 1$ , such that  $\delta'^*(s, w) \in F' \Leftrightarrow \delta'^*(t, w) \notin F'$ .

Let us split  $w$  as  $w = uv$  such that  $|v| = i$ , which is exactly one symbol longer than the longest minimization word of  $A$ . We can formulate the following statement:

There must exist  $p, q \in Q'$  such that  $\delta'^*(p, v) \in F' \Leftrightarrow \delta'^*(q, v) \notin F'$ . (3.1)

**Gregor:** hidden formulations here

We can therefore state, that  $\neg(p \in Q \wedge q \in Q)$ , because else  $\mathcal{D}(A)$  would be higher than  $i$  too. So at least one of  $p, q$  must be in  $Q' \setminus Q$  which is exactly  $\{q_d^1, \dots, q_d^n\}$ .

- Every  $q_d^k$  is  $d_{A'}$ -equivalent to a  $q \in Q$
- In every case,  $p, q$  can be  $d_{A'}$ -exchanged s.t.  $p, q \in Q$
- But that's contradictory to  $\mathcal{D}(A) = n$ , because  $p, q$  belong to a minimization word  $w = n - 1$

□

**Gregor:** Old proof for one  $q_d$

**Gregor:** hidden old 'systematic study of how to extend minimal DFAs'

## 3.2 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.

```

1: function ADDUNREACHABLESTATES ( $A, u$ )
2:   for  $u$  times do
3:      $q \leftarrow \max Q + 1$ 
4:      $Q \leftarrow Q \cup \{q\}$ 
5:      $R \leftarrow$  random chosen sample of  $|\Sigma|$  states from  $Q \setminus \{q\}$ 
6:     for  $\sigma$  in  $\Sigma$  do
7:        $q' \in R$ 
8:        $R \leftarrow R \setminus \{q'\}$ 
9:        $\delta \leftarrow \delta \cup \{((q, \sigma), q')\}$ 
10:  return  $A$ 

```

We have to ensure, that this algorithm does not induce changes in the language.

**Lemma 3.** *Adding unreachable states to a DFA does not change its language.*

*Proof.* Remember that the language of a DFA  $A = (Q, \Sigma, \delta, s, F)$  is defined as  $L(A) = \{ w \mid w \in \Sigma^* \}$ . For any unreachable state  $q$  there exists no word  $v \in \Sigma^*$  such that  $\delta^*(s, v) = q$ . Thus such a state cannot be the cause for any word to be in  $L(A)$ . □

The question whether adding unreachable states to a DFA changes  $\mathcal{D}$ -value is irrelevant. This is because in the context of the minimization algorithm, unreachable states are eliminated before the MINIMIZATIONMARK-algorithm is applied on the task DFA.

# Chapter 4

## Notes on the implementation

- what is implemented
- maybe module, functions overview
- maybe speedtest/heatmap results



# Chapter 5

## Conclusion

What happens, if we change start and accepting states?

What happens, if we add transitions only?

dfa specific planarity test?

use planarity test information for better drawing?

# Bibliography

- [1] 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.
- [2] John Hopcroft and Robert Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, October 1974.
- [3] 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.
- [4] William Kocay. The hopcroft-tarjan planarity algorithm. October 1993.
- [5] Uwe Schöning. *Theoretische Informatik - kurzgefasst*. Spektrum, Akad. Verl, Heidelberg Berlin, 2001.