Generation of DFA Minimization Problems

Gregor Hans Christian Sönnichsen

February 10, 2020

Abstract

Abstract...

Generierung von DFA Minimierungsproblemen
-
Zusammenfassung

Zusammenfassung. . .

Table of Contents

Chapter 1

Introduction

- study computer science
- theoretical informatics
- automata theory
- value of this theory
- typical topics (minimization), why typical
- sketch situation
- why

  - science

  - increased clarification which kind of tasks can be generated, and where difficulties of those tasks lie

  automation allows

  - freeing time for other things, e.g. research, helping students face-to-face, designing the whole exercise sheet

  - generation of tasks which lie in a well-defined range

  - increased predictability and consistency of the generated task properties, which can be adjusted accurately through various parameters

  - saves human operators from the generating task which involves monotonous work

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.

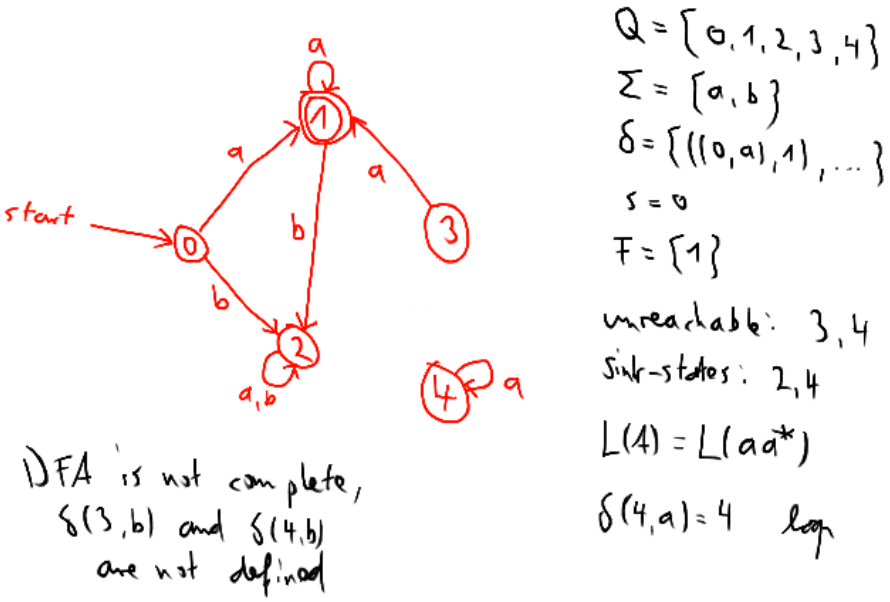

Figure 1.1: An example DFA and its properties.

1.1.1 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) [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 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 that DFA is defined as $L(A) = \{\ w \mid \delta^*(w) \in F\ \}$ [3, 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 [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: $\forall q \in Q\colon \forall \sigma \in \Sigma\colon \exists p \in Q\colon \delta(q, \sigma) = p$. Note, that every incomplete DFA can be converted to a complete one by adding a so called *dead state* [3, p. 67]. The resulting automaton has the same language.

1.1.2 Minimal DFAs

This section closely follows [6, 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 Σ is defined as follows:

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

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 Σ 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 Practical 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 *practical isomorph*, iff:

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

Note that practical isomorphy between two DFAs A_1, A_2 does not imply $L(A_1) = L(A_2)$. This would be given, if $\Sigma_1 = \Sigma_2$ were the case (see [6, p. 45]). However the language of such DFAs is equivalent except for an exchange of alphabet symbols:

$$\{\ \phi(\sigma_0) \dots \phi(\sigma_n) \mid \sigma_0 \dots \sigma_n \in L(A_1)\ \} = L(A_2)$$

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

1.1.4 Equivalent and distinguishable state pairs

Definition 2 (Equivalent and Distinguishable State Pairs)**.** [3, p. 154] A state pair $q_1, q_2 \in Q$ of a DFA $A = (Q, \Sigma, \delta, s, F)$ is called *equivalent*, iff $\sim_A\ (q_1, q_2)$ is true, whereas

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

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

1.1.5 The minimization algorithm

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

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

```
1: function ComUnreachables(A)
2:     U ← Q \ {s}                                   ▷ undiscovered states
3:     O ← {s}                                       ▷ observed states
4:     D ← {}                                        ▷ discovered states
5:     while |O| > 0 do
6:         N ← { p | ∃q ∈ O σ ∈ Σ: δ(q, σ) = p ∧ p ∉ O ∪ D }
7:         U ← U \ N
8:         D ← D ∪ O
9:         O ← N
10:    return U
```

2. Remove all unreachable states and their transitions.

```
1: function RemUnreachables(A, U)
2:     δ' ← δ \ { ((q, σ), p) | q ∈ U ∨ p ∈ U }
3:     return (Q \ U, Σ, δ', s, F \ U)
```

3. Compute all equivalent state pairs ($\sim_A$).

```
1: function ComEquivPairs(A)
2:     M ← {(p, q), (q, p) | p ∈ F, q ∉ F}
3:     do
4:         M' ← {(p, q) | (p, q) ∉ M ∧ ∃σ ∈ Σ: (δ(p, σ), δ(q, σ)) ∈ M}
5:         M ← M ∪ M'
6:     while M' ≠ ∅
7:     return Q² \ M
```

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

4. Merge all equivalent state pairs, which are exactly those in $\sim_A$.

```
1: function RemEquivPairs(A, ∼A)        ▷ [·]∼A shall be abbreviated [·]
2:     Q_E ← ∅
3:     δ_E ← ∅
```

```
    F_E ← ∅
    for q in Q do
        Add [q] to Q_E
        for σ in Σ do
            δ_E([q], σ) = [δ(q, σ)]
        if q ∈ F then
            Add [q] to F_E
    return (Q_E, Σ, δ_E, [s], F_E)
```

Note that RemEquivPairs creates complete automatons.

Theorem 2. *The minimization algorithm computes a minimal DFA to its input DFA.*

The definition of this DFA minimization algorithm is inspired by Schöning [6, p. 46], Hopcroft [3, p. 161] and Högberg [4, p. 10].

1.1.6 m-ComEquivPairs.

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

```
function m-ComEquivPairs(A)
    i ← 0
    m(0) ← {(p, q), (q, p) | p ∈ F, q ∉ F}
    do
        i ← i + 1
        m(i) ← {(p, q) | (p, q) ∉ ⋃m(·) ∧ ∃σ ∈ Σ: (δ(p, σ), δ(q, σ)) ∈ m(i − 1)}
    while m(i) ≠ ∅
    return ⋃m(·)
```

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 ComEquivPairs 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?

1.1.7 Another equivalence automaton

Consider step 3 and 4 of MinimizeDFA. A view on these algorithms reveal, that they are essentially collapsing each $\sim$-equivalence class of a DFA A into one state.

Definition 3 ($\sim_A$-equivalence automaton)**.** We will call the automaton $E_A = (Q_E, \Sigma_E, \delta_E, s_E, F_E)$ created by ComEquivPairs and RemEquivPairs(A) the *$\sim_A$-equivalence automaton* of A. It has the following properties:

- $Q_E = \{\ [q] \mid q \in Q\ \}$
- $\Sigma_E = \Sigma$

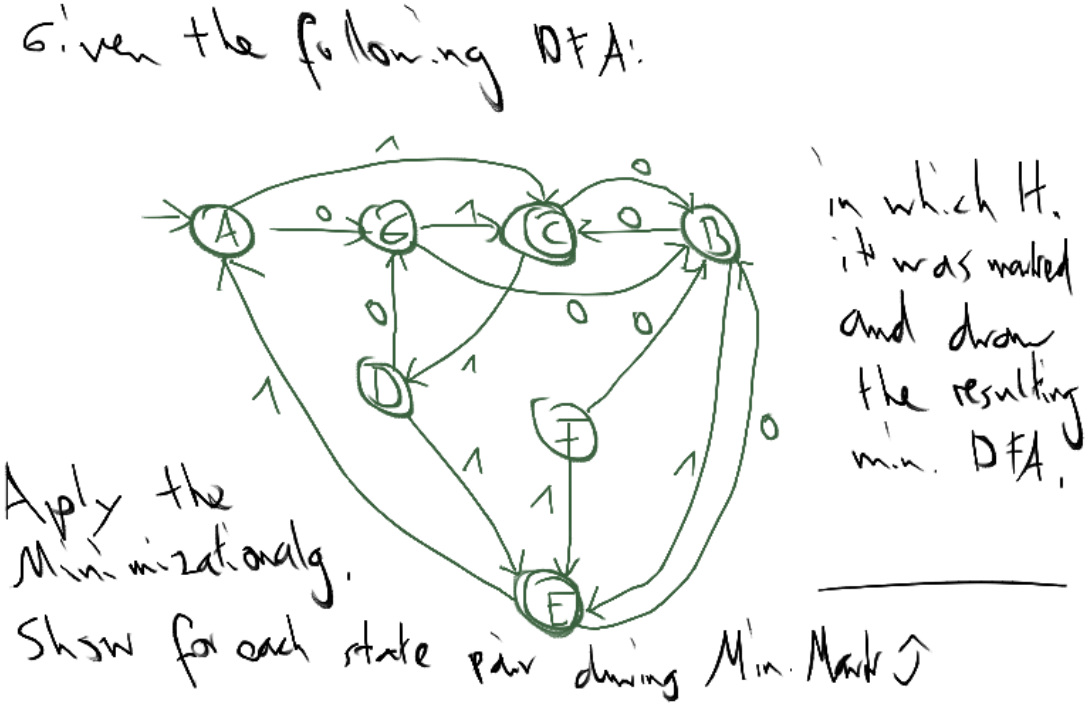

Figure 1.2: An example DFA minimization task.

- $\delta_E([q], \sigma) = [\delta(q, \sigma)], \forall q \in Q, \sigma \in \Sigma$
- $s_E = [s]_\sim$
- $F_E = \{\ [q] \mid q \in Q\ \}$

Note, that we know by theorem 2 that E_A is guaranteed to be a minimal automaton to A, if A has no unreachable states.

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

Gregor: search for typical task in standard text books

Figures 1.2 and 1.3 show such a task and solution. The students are confronted with a *task DFA* A_{task}. Firstly, unreachable states have to be eliminated, we then gain the *reachable DFA* A_{re}. Secondly equivalent state pairs of A_{re} are merged such that the minimal *solution DFA* A_{sol} is found. The table T displayed in figure 1.3 is nothing else but a visualization of the function m, whereas $T(q_0, q_1) = i \Leftrightarrow (q_0, q_1) \in m(i)$.

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

- $A_{re} = \textsc{RemUnreachables}(A_{task}, \textsc{ComUnreachables}(A_{task}))$ and

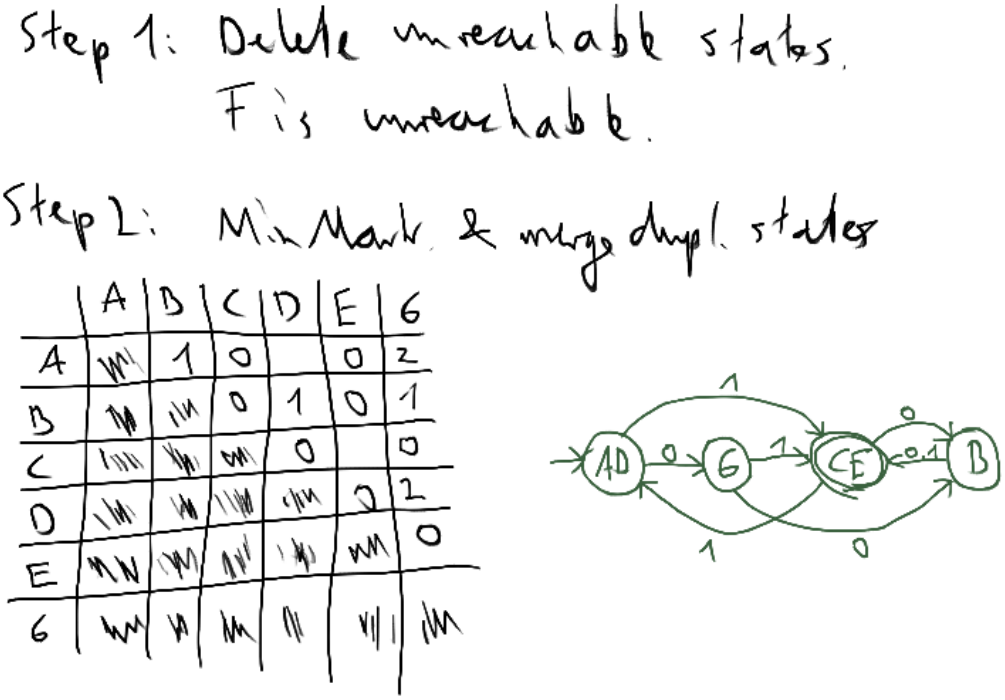

Figure 1.3: Solution to the DFA minimization task in fig. 1.2.

- $A_{sol} = \text{RemEquivPairs}(A_{re}, \text{ComEquivPairs}(A_{re}))$

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

Gregor: How to define 'already found DFA sol' as requirement. Final states argument missing.

1.2.2 Difficulty adjustment possibilities

Concerning the execution of MinimizeDFA we find that its difficulty can be classified through various classification numbers.

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

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

Number of states. To control the number of states in A_{task}, A_{re} and A_{sol}, we will introduce three parameters: $\mathfrak{e}, \mathfrak{r}, \mathfrak{u} \in \mathbb{N}$. These parameters get their meaning by the following equations:

$$\begin{aligned} |Q_{sol}| &= \mathfrak{e} \\ |Q_{re}| &= \mathfrak{e} + \mathfrak{r} \\ |Q_{task}| &= \mathfrak{e} + \mathfrak{r} + \mathfrak{u} \end{aligned}$$

It is sensible to have $\mathfrak{u} > 1, \mathfrak{r} > 1$, such that RemUnreachables and RemEquivPairs will not be skipped. To not make the task trivial, $\mathfrak{e} > 2$ is sensible. An exercise instructor will find it useful, to control exactly how big $\mathfrak{u}$, $\mathfrak{r}$ and $\mathfrak{e}$ are: The higher $\mathfrak{u}, \mathfrak{r}$, the more states have to be eliminated and merged. The higher $\mathfrak{e} + \mathfrak{r}$, the more state pairs have to be checked during ComEquivPairs.

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

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

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

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

1.2.3 Summary of found criteria

Accepted general criteria:

-> $L(A_{sol}) = L(A_{re}) = L(A_{task})$

-> $\mathcal{D}(A_{sol}) = \mathcal{D}(A_{re}) = \mathcal{D}(A_{task})$

Accepted solution DFA criteria:

-> has to be minimal, complete

-> number of essential states

-> number of ComEquivPairs iterations ($\mathcal{D}(A_{sol})$)

-> alphabet size

-> number of accepting states

-> planarity

-> A_{sol} is new

Definition 4 (New DFAs)**.** A DFA A_{sol} is *new* if it is not practically isomorph to any previously generated solution DFA.

Accepted reachable DFA criteria:

-> has to be complete

-> number of unreachable states

-> planarity (can be checked in $O(|Q_{task}|)$)

Accepted task DFA criteria:

-> number of redundant states

-> planarity

-> completeness

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 unreachable and redundant 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}. We know that adding redundant and unreachable states does not change $L(A_{task})$ in comparison to A_{sol}, else MinimizeDFA would not work (a minimal DFA has in particular the same language as the original DFA). However we must ensure, that adding those states does not change $\mathcal{D}$. Since unreachable states are eliminated before ComEquivPairs is applied, we need only to ensure, that adding redundant states does not change the $\mathcal{D}$-value. We will do this during the discussion of step 2, more specifically in section 3.1.4.

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

1: **function** GenerateDFAMinimizationProblem($\mathfrak{e}, a, f, m_{min}, m_{max}, p_1, p_2, \mathfrak{r}, \mathfrak{u}$)
2: $A_{sol} \leftarrow$ GenerateNewMinimalDFA($\mathfrak{e}, a, f, m_{min}, m_{max}, p_1$)
3: $A_{task} \leftarrow$ ExtendMinimalDFA($A_{sol}, p_2, \mathfrak{r}, \mathfrak{u}$)
4: **return** A_{sol}, A_{task}

Chapter 2

Generating minimal DFAs

We seek algorithms for generation of minimal DFAs that fulfilling the conditions defined in the requirements analysis section 1.2.3. We formally subsume these conditions via the GenerateNewMinimalDFA-problem:

Definition 5 (GenerateNewMinimalDFA)**.**
Given:

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

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

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

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

Note that the presented algorithms will not be able to compute all of $\mathcal{A}_{min}$ since we are going to exclude minimal DFAs that are practical isomorph to already found ones.

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 new. 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 which are in particular finite.

Gregor: How much smaller? Why now finite?

1: **function** BUILDNEWMINIMALDFA-1 $(\mathfrak{e}, a, f, m_{min}, m_{max} \in \mathbb{N}, p \in \{0,1\})$
2: **while** True **do**
3: generate DFA A_{test} with $|Q|, |\Sigma|, |F|$ matching $\mathfrak{e}, a, f$
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} is not new **then**
9: **continue**
10: **return** A_{test}

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

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 isomorphy test. However it is sufficient to ensure, that no redundant or unreachable states exist. **Gregor:** minimality planarity complete under isomorphy

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

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

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

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

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

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*[1] has been used, which implements the Hopcroft-Tarjan planarity algorithm. More information on this can be found for example in this [5] introduction from William Kocay. The original paper describing the algorithm is [2].

2.1.3 Ensuring A_{test} is new

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

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

With this scheme we can fetch once all DFAs matching the search parameters. Thus we need not fetch all previously found DFAs every time, but only those that are relevant. Afterwards we must only check whether any practical 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 $(\mathfrak{e}, a, f, m_{min}, m_{max}, p)$
2: $l \leftarrow$ all DFAs in DB1 matching $\mathfrak{e}, a, f, m_{min}, m_{max}, p$
3: **while** True **do**
4: ...
5: **if** A_{test} is practical 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 **??** tells us, that the states names are irrelevant for the minimality of a DFA, therefore we will give our generated DFAs simply the states $e_1, \ldots, e_{\mathfrak{e}}$. 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 $e_1 \in Q$, since we can apply an isomorphism to every DFA, such that its start state is renamed to e_1.

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

The remaining elements that need to be defined are δ 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.

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 $e \in Q$ and $\sigma \in \Sigma$ we can randomly choose an end state from Q.

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

```
1: function BuildNewMinimalDFA-3a (𝔢, a, f, m_min, m_max, p)
2:     l ← all DFAs in DB1 matching 𝔢, a, f, m_min, m_max, p
3:     Q ← {e_1, ..., e_𝔢}
4:     Σ ← {σ_1, ..., σ_a}
5:     while True do
6:         δ ← ∅
7:         for e in Q do
8:             for σ in Σ do
9:                 e' ← random chosen state from Q
10:                δ ← δ ∪ {((e, σ), e')}
11:        s ← e_1
12:        F ← random sample of f states from Q
13:        A_test ← (Q, Σ, δ, s, F)
14:        if A_test not minimal or not m_min ≤ D(A_test) ≤ 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 δ, we could just enumerate through all possible final state sets and transition functions.

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

Again we will call our states and symbols w.l.o.g. $e_1, \ldots, e_{\mathfrak{e}}$ resp. $\sigma_1, \ldots, \sigma_a$. We will represent the state of an enumeration with two fields F_F and F_δ. The first field shall have $\mathfrak{e}$ Bits, whereas Bit $F_F[i] \in [0, 1]$ represents the information, whether $\mathfrak{e}_i$ is a final state or not. The second field shall have $\mathfrak{e} * a$ entries containing state indices, such that entry $F_\delta[i * a + j] = k, k \in [1, \mathfrak{e}]$ says, that $\delta(e_i, \sigma_j) = e_k$. These semantics are illustrated in figure 2.1.

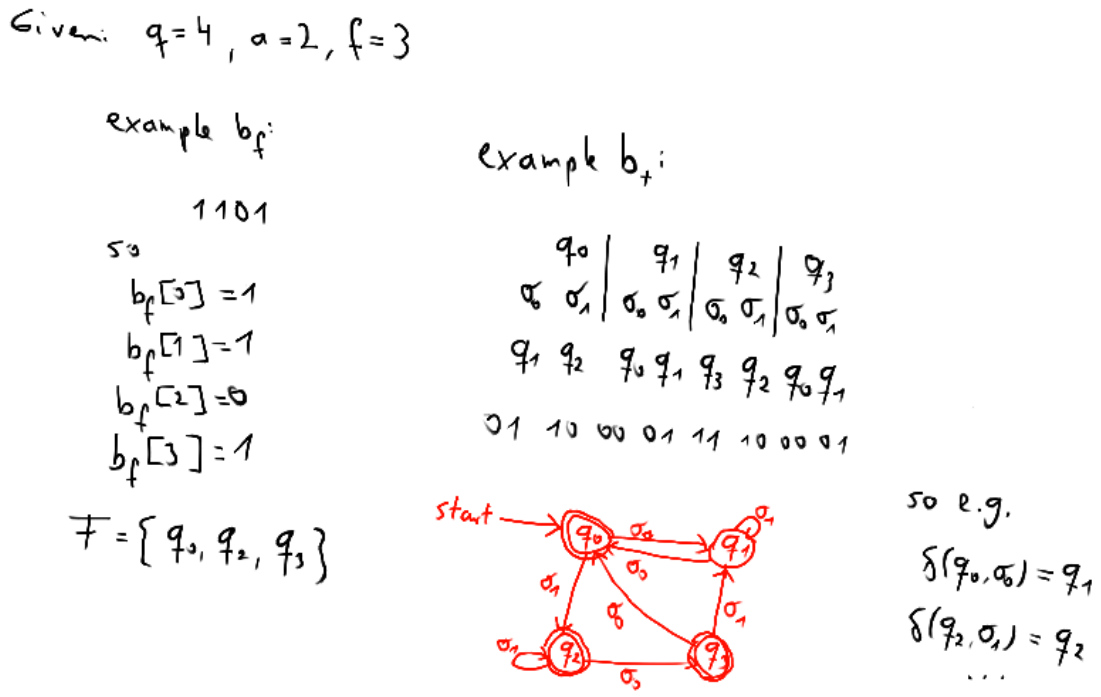

Figure 2.1: Example for two possible configurations of the fields F_F and F_δ given $\mathfrak{e}, a$ and f. Below the corresponding DFA is drawn.

Given an enumeration state b_f, b_t and $\mathfrak{e}, a, f$ 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 $\log_2(\mathfrak{e})$-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 happens at 11111...

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

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:

function DFAfromEnumProgress $(F_F, F_\delta, \mathfrak{e}, a, f)$
  $Q \leftarrow \{e_1, \dots, e_\mathfrak{e}\}$
  $\Sigma \leftarrow \{\sigma_1, \dots, \sigma_a\}$
  $\delta \leftarrow \emptyset$
  for i **in** $[1, \dots, \mathfrak{e}]$ **do**
    for j **in** $[1, \dots, a]$ **do**

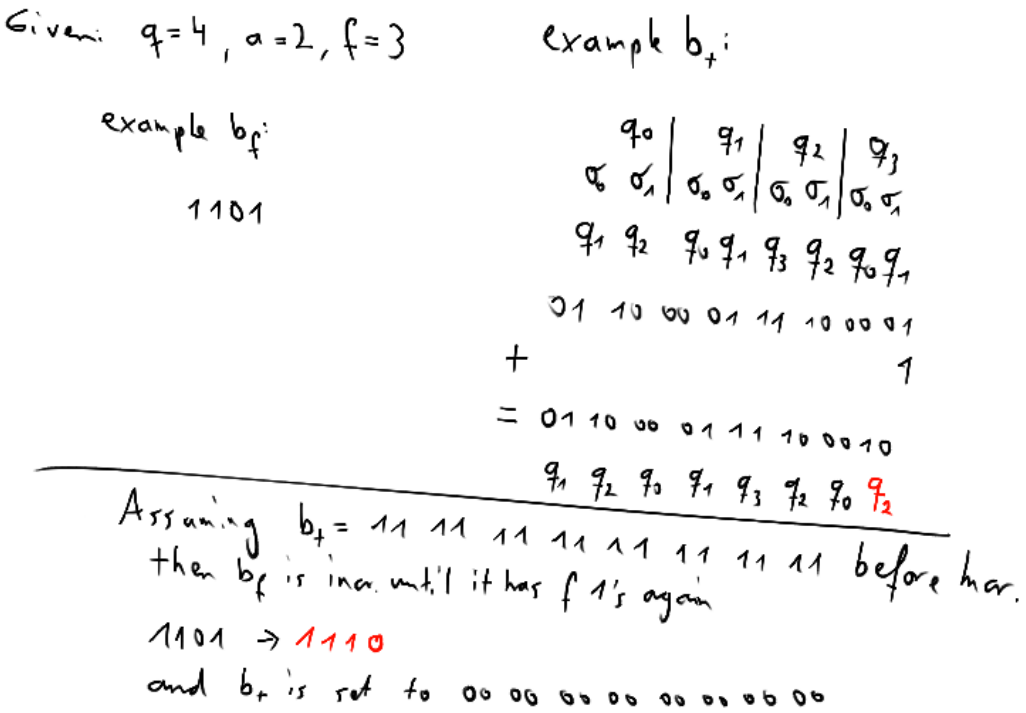

Figure 2.2: The upper half shows how a F_δ-increment results in a change in the resulting DFAs transition function: $\delta(e_3, \sigma_1) = e_1$ becomes $\delta(e_3, \sigma_1) = e_2$. The lower half shows what happens, if F_δ has reached its end.

7: $\quad\quad\quad \delta(e_i, \sigma_j) = e_{F_\delta[i*a+j]}$

8: $\quad s \leftarrow e_1$

9: $\quad F \leftarrow \{\ e_i \mid i \in [1, \ldots, \mathfrak{e}] \wedge F_F[i] = 1\ \}$

10: $\quad$ **return** $(Q, \Sigma, \delta, s, F)$

The initial field values are each time $0 \ldots 0$. Note how construction and use of these fields results in DFAs with correct alphabet size and number of (final) states. An enumeration can finish either because a matching DFA has been found or all DFAs have been enumerated.

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

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

We reduce the enumeration room for each calculation.

1: **function** BuildNewMinimalDFA-3b $(\mathfrak{e}, a, f, m_{min}, m_{max}, p)$

2: $\quad l \leftarrow$ all DFAs in DB1 matching $\mathfrak{e}, a, f, m_{min}, m_{max}, p$

3: $\quad F_F, F_\delta \leftarrow$ load enumeration progress for $\mathfrak{e}, a, f, p$ from DB2

4: $\quad$ **while** True **do**

5: $\quad\quad$ **if** F_F, F_δ is finished **then**

```
6:          save $F_F, F_\delta$
7:          return $\perp$
8:      $A_{test} \leftarrow$ next DFA based on $F_F, F_\delta$
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 $F_F, F_\delta$ 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

2.1.7 Related Research

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

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

Chapter 3

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 1.2.3):

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

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

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

- $Q_{task} = Q_{sol} \cup \{r_1, \ldots, r_{\mathfrak{r}}, u_1, \ldots, u_{\mathfrak{u}}\}$
- $r_1, \ldots, r_{\mathfrak{r}}$ *redundant*
- $u_1, \ldots, u_{\mathfrak{u}}$ *unreachable*
- $\Sigma_{task} = \Sigma_{sol}$, $s_{task} = s_{sol}$, $F_{task} \subseteq F_{sol}$
- A_{task} *being planar iff* $p = 1$
- A_{task} *being complete iff* $c = 1$
- A_{sol} *being isomorph to* MinimizeDFA(A_{task})

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

3.1 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 $r_1, \ldots, r_{\mathfrak{r}}$ to a DFA A_{sol}, gaining A_{re} with $Q_{re} = Q_{sol} \cup \{r_1, \ldots, r_{\mathfrak{r}}\}$, such that each of these states is equivalent to a state in A_{re}. Note that, for reasons of clarity, we are going to abbreviate from now on $A_{re} = A$, $Q_{re} = Q$, $\sim_{A_{re}} = \sim_A$ etc.

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

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

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

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

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

3.1.1 Adding outgoing transitions

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

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

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

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

3.1.2 Adding ingoing transitions

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

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

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

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

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

These finding lead us to a general requirement regarding the choice of a state e for an r_i: The equivalence class of any e has to contain at least one state with at least 2 ingoing transitions (see fig. 3.1). We establish the following notion to pin down this restriction:

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

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

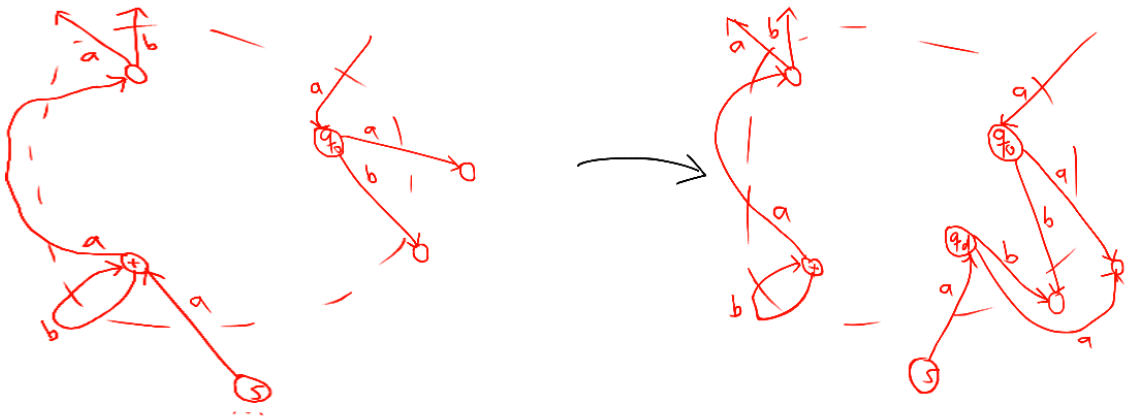

Figure 3.1: If an equivalence class (here denoted by the states in the dashed area) contains a state with 2 or more ingoing transitions (in this case p), then a state equivalent to any of the classes states may be added. Here r is equivalent to o and is "stealing" the ingoing transition $\delta(q, a)$ from p.

3.1.3 The algorithm

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

3.1.4 Adding redundant states does not change $\mathcal{D}$

To prove this statement, we will prove two minor propositions first. In this context we will call a word w *distinguishing word of* p, q, iff $distinguishes(w, q_0, q_1)$ whereas $distinguishes_A(w, q_0, q_1) \Leftrightarrow (\delta^*(p, w) \in F \Leftrightarrow \delta^*(q, w) \notin F)$.

Lemma 1 (Semantics of $(p, q) \in m(n)$)**.** *In the context of* ComEquivPairs *the following is true:*

$$(p, q) \in m(n) \Longleftrightarrow \exists w \in \Sigma^* \colon (|w| = n \ \wedge distinguishes_A(w, p, q))$$

Proof. See TI-Lecture ch. 4 "Minimization" p. 18. □

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

$$\mathcal{D}(A) = \ n \Rightarrow \quad n = \max_{n \in \mathbb{N}} \ \exists p, q \in Q \ \ \exists w \in \Sigma^* \colon \ (|w| = n - 1 \ \wedge distinguishes_A(w, p, q))$$

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

1. $\forall i \in [0, n-1] \colon m(i) \neq \emptyset$
2. $\forall i \geq n \colon m(n) = \emptyset$ **Gregor:** fix this

m-ComEquivPairs(A) terminates iff $m(i) = \emptyset$. If the first point would not hold, then the algorithm would have stopped before.

Since the algorithm did n iterations, the internal variable i must be n at the end of the last iteration. The terminating condition is $m(i) \neq \emptyset$; thus follows the second point.

We prove that then there exists a distinguishing word of length $n - 1$, but the existence of a longer word leads to a contradiction. Recall lemma 1:

$$(p, q) \in m(n) \Longleftrightarrow \exists w \in \Sigma^* \colon (|w| = n \ \wedge distinguishes(w, p, q))$$

Following this lemma and point 1, we can deduce that there exists at least one distinguishing word w with $|w| = n - 1$ for some $p, q \in Q$.

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

Theorem 3. *Creating equivalent state pairs in a minimal DFA A does not increase the number of iterations when the* ComEquivPairs*-algorithm is applied on it.*

Proof. Proof per contradiction.
Let us assume there were $\mathfrak{r}$ states $r_1, \ldots, r_\mathfrak{r}$ added to the given minimal DFA $A = (Q, \Sigma, \delta, s, F)$ resulting in a DFA $A' = (Q', \Sigma, \delta', s, F')$ such that $\mathcal{D}(A) < \mathcal{D}(A')$. Concerning A' we can observe the following:

- $Q' = Q \cup \{r_1, \ldots, r_\mathfrak{r}\}$

- W.l.o.g. $\forall i \in [1, \mathfrak{r}]\colon\ \exists e \in Q_{sol}\colon\ r_i \sim_{A'} e$

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

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

According to this lemma there are words w with $|w| = i-1$ and w' with $|w'| = j-1$ which are the longest distinguishing words for some states of A resp. some states q', p' of A'.

Recall now

Lemma 3.

$$distinguishes_A(w, p, q) \wedge q \sim_A q' \Rightarrow distinguishes_A(w, p, q')$$

Case 1 - p', q' both in Q

Since the longest distinguishing word any state pair $p, q \in Q$ can have is guaranteed shorter than w', we know that no $p, q \in Q$ can have w' as distinguishing word. ↯

Case 2 - one of p', q' in Q, one in $Q' \setminus Q$

W.l.o.g. $p' \in Q, q' \in Q' \setminus Q$. We then know, that $q' \in \{r_1, \ldots, r_{\mathfrak{r}}\}$. This implies that $q' \sim_{A'} e$ for an $e \in Q$. By lemma 3 $distinguishes_A(w', p', e)$. But then two states from Q would have w' as distinguishing word which is contradictory, see above. ↯

Case 3 - p', q' both in $Q' \setminus Q$

Analogue to case 2, we can find states $e_1, e_2 \in Q$ equivalent to p', q'. Then again two states from Q would have w' as distinguishing word. ↯

Thus we have led our assumption to contradiction in every case, which finishes the proof.

□

Gregor: Old proof for one q_r

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 ← max Q + 1
4:         Q ← Q ∪ {q}
```

5: $R \leftarrow$ random chosen sample of $|\Sigma|$ states from $Q \setminus \{q\}$
6: **for** σ **in** Σ **do**
7: $q' \in R$
8: $R \leftarrow R \setminus \{q'\}$
9: $\delta \leftarrow \delta \cup \{((q, \sigma), q')\}$
10: **return** A

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] 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.

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

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

Erklärung

Hiermit versichere ich, Gregor Hans Christian Sönnichsen, dass ich die vorliegende Arbeit selbständig angefertigt und ohne fremde Hilfe verfasst habe, keine außer den von mir angegebenen Hilfsmitteln und Quellen dazu verwendet habe und die den benutzten Werken inhaltlich oder wörtlich entnommenen Stellen als solche kenntlich gemacht habe.

Bayreuth, den 8. Februar 2020.

Gregor Hans Christian Sönnichsen