Generation of DFA Minimization Problems

Gregor H. C. Sönnichsen

February 7, 2020

Contents

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*, 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].

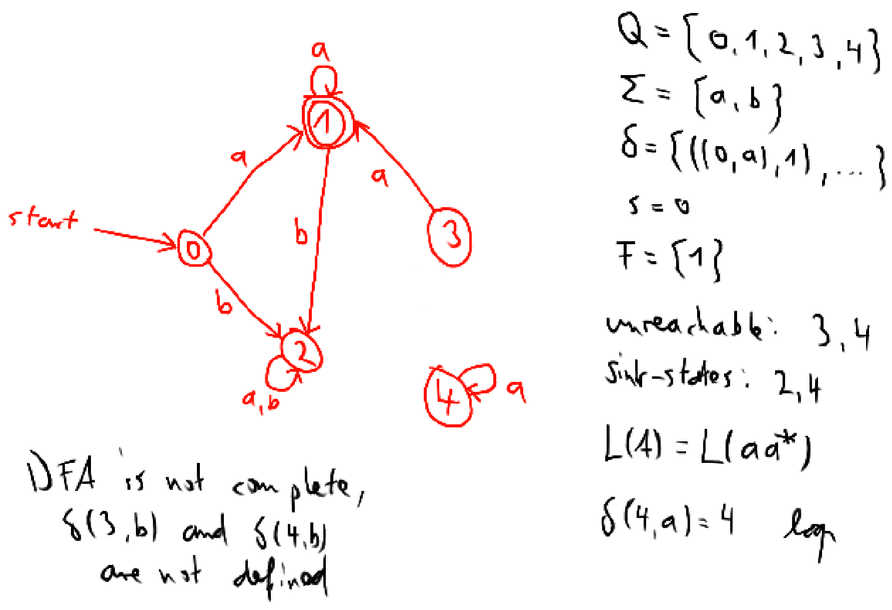

Figure 1.1: An example DFA and its properties.

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 [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 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:
  $\pi(s_1) = s_2$
  $\forall q \in Q_1 \colon (q \in F_1 \iff \pi(q) \in F_2)$
  $\forall q \in Q_1 \colon \forall \sigma \in \Sigma_1 \colon \pi(\delta_1(q, \sigma)) = \delta_2(\pi(q), \phi(\sigma)))$

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 [5, p. 45]). However the language of such DFAs is equivalent except for an exchange of alphabet symbols:

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

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 $q_0, q_1 \in Q$ of a DFA $A = (Q, \Sigma, \delta, s, F)$ is called *equivalent*, iff $\sim_A (q_0, q_1)$ is true, whereas

$$q_0 \ \sim_A \ q_1 \ \Leftrightarrow_{def} \ \forall z \in \Sigma^* \colon \ (\delta^*(q_0, z) \in F \Leftrightarrow \delta^*(q_1, z) \in F)$$

If $(q_0, q_1) \notin \sim_A$, then q_0 and q_1 are called a *distinguishable* state pair.

Note that the relation $\sim_A$ is indeed an equivalence relation.

- equivalent state pairs
- equivalent states
- distinguishable state pairs
- distinguishable states

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 for example.

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

2. Remove all unreachable states and their transitions.

```
function RemoveUnreachables($A, U$)
    for $q$ in $U$ do
        if $q \in F$ then
            $F \leftarrow F\ \{q\}$
        $\delta \leftarrow \delta \setminus \{\ ((q_1,\sigma), q_2) \in \delta \mid q_1 = q \vee q_2 = q\ \}$
    return $A$
```

3. Compute all distinguishable state pairs ($\neg \sim_A (p,q)$).

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

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

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

```
function RemoveEquivalentPairs($A, \neg \sim_A$)
    $\sim_A \leftarrow Q^2 \setminus \neg \sim_A$
    while $(p,q) \in \sim_A$ do
        $\sim_A \leftarrow \sim_A \setminus \{(p,q)\}$
        if $p = q$ then
            continue

        $Q \leftarrow Q \setminus \{q\}$
        if $q \in F$ then
```

$F \leftarrow F \setminus \{q\}$
for $((q_0, \sigma), q_1)$ **in** δ **do**
  if $q_0 = q$ **then**
    $q_0 \leftarrow p$
  if $q_1 = q$ **then**
    $q_1 \leftarrow p$

for (q_0, q_1) **in** $\sim_A$ **do**
  if $q_0 = q$ **then**
    $q_0 \leftarrow p$
  if $q_1 = q$ **then**
    $q_1 \leftarrow p$
return A

Note that RemoveEquivalentPairs preserves completeness, since it does only remove transitions from those state, that are removed anyway from the automaton. **Gregor:** RemoveEquivalentPairs constructs possibly non-det. DFAs on its way. Write it more explicit. Probably someone has?

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

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

m-ComputeDistinguishablePairs. When looking at ComputeDistinguishablePairs, 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-ComputeDistinguishablePairs(A)
  $i \leftarrow 0$
  $m(0) \leftarrow \{(p, q), (q, p) \mid p \in F, q \notin F\}$
  do
    $i \leftarrow i + 1$
    $m(i) \leftarrow \{(p, q) \mid (p, q) \notin \bigcup m(\cdot) \land \exists \sigma \in \Sigma \colon (\delta(p, \sigma), \delta(q, \sigma)) \in m(i-1)\}$
  while $m(i) \neq \emptyset$
  return $\bigcup m(\cdot)$

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

We will denote the number of iterations done by ComputeDistinguishablePairs 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?

1.1.6 Essential and redundant states

When looking at ComputeDistinguishablePairs and RemoveEquivalentPairs one furthermore notes, that they essentially

1. compute the equivalence classes of $\sim_A$ (by exploring for each state pair whether its equivalent)

   $eq\_classes(\sim_A) = \{[q]_{\sim_A} | q \in Q\} = \{C_0, \ldots, C_n\}$

2. choose one state e_i of each equivalence class C_i and merge all other states towards it (RemoveEquivalentPairs never creates new states, but transfers transitions)

These dedicated states $e_0, \ldots, e_n$ then correspond exactly to the states of the equivalence automaton - each state represents one equivalence class, and every equivalence class is represented by one state.

Since MinimizeDFA can be applied to any DFA, we can be sure that there exist states $e_0, \ldots, e_n$ in every automaton A, which would remain as set of states for the automaton MinimizeDFA(A). We shall name these states *essential* states. All states that will not be part of the by MinimizeDFA minimized DFA will be called *redundant* states.

Gregor: Example: In 1.2 and 1.3 the state pairs $(A, D), (C, E)$ are equivalent and all others distinguishable. The states A, G, C, B are essential, for they show up in the minimized automaton. The states D, E are therefore redundant.

As a consequence saying that RemoveEquivalentPairs *merges equivalent state pairs* is equivalent to saying it *removes redundant 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

- search for typical test in text standard work books

Figures and 1.3 show such a task and solution. In a DFA minimization task (fig. 1.2) students are confronted with a *task DFA* A_{task}, that is to be minimized by eliminating unreachable states (thus gaining the *intermediate DFA* A_{inter}) and merging equivalent state pairs towards a *solution DFA* A_{sol} using the minimization algorithm. The table T displayed in the solution is nothing else but a visualization of the function m, whereas $T(q_0, q_1) = i \Leftrightarrow (q_0, q_1) \in m(i)$.

There are some formal statements and requirements. Firstly, we can state that

- A_{inter} = RemoveUnreachables(A_{task}, ComputeUnreachables(A_{task})) and
- A_{sol} = RemoveEquivalentPairs(A_{inter}, ComputeDistinguishablePairs(A_{inter}))

Therefore A_{sol} has to be minimal regarding A_{inter} and A_{task}. Secondly the languages of A_{task}, A_{inter} and A_{sol} must be equal. We know that ComputeDistinguishablePairs requires A_{inter} to be complete and that RemoveEquivalentPairs preserves completeness, so A_{sol} is complete too.

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

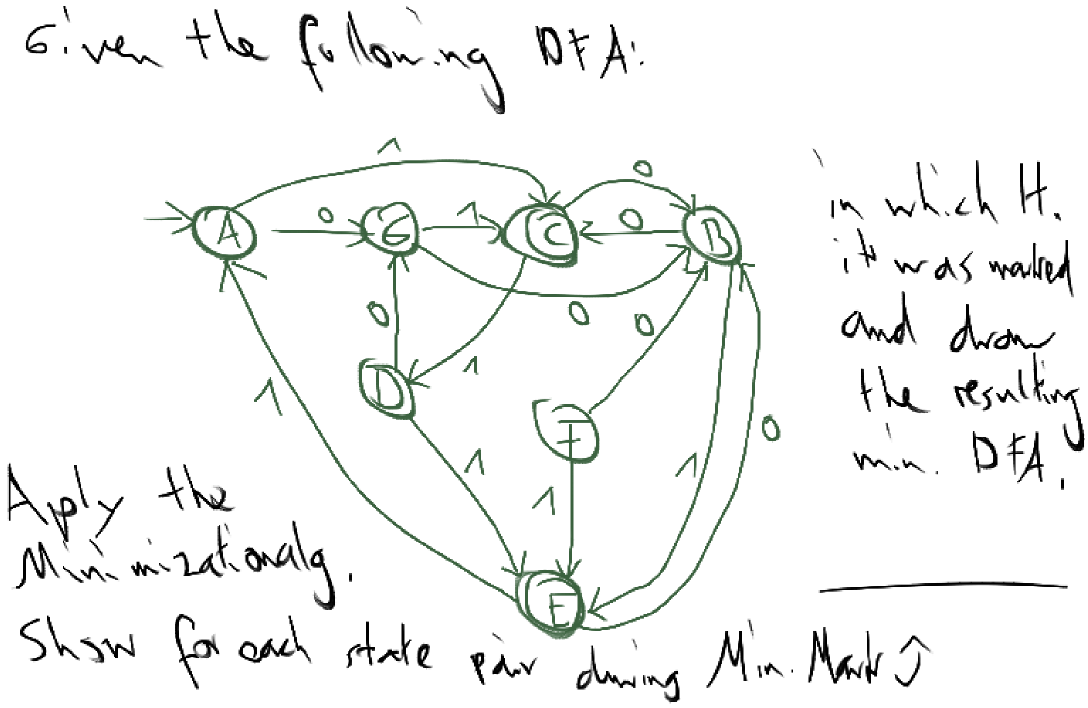

Figure 1.2: An example DFA minimization task.

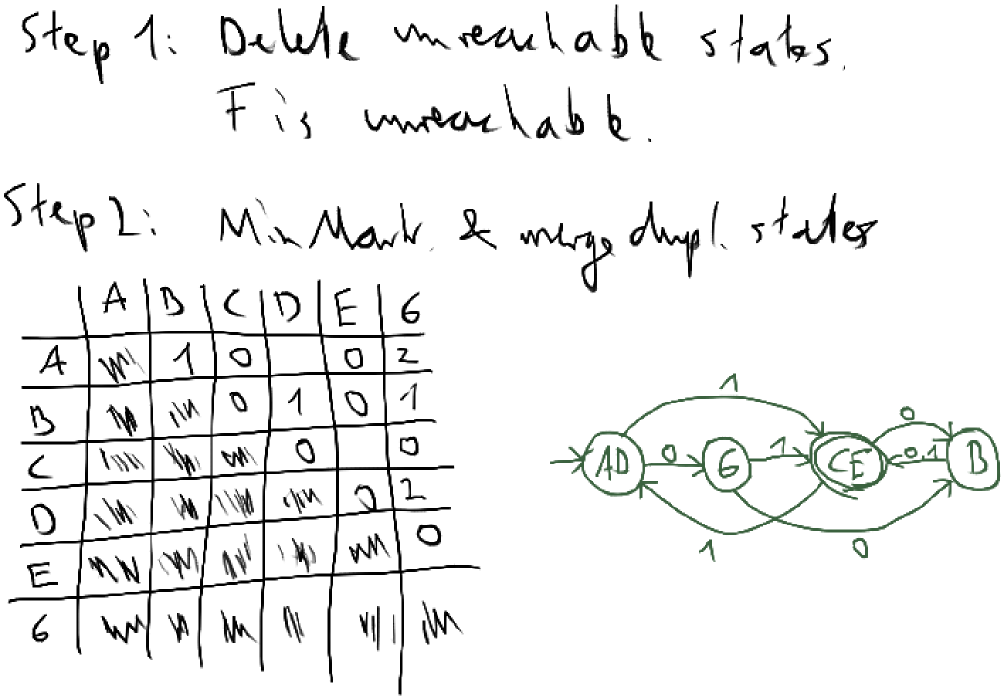

Figure 1.3: Solution to the DFA minimization task in fig. 1.2.

ComputeDistinguishablePairs-depth ($\mathcal{D}(A_{task})$). Consider the computation of the sets $m(i)$ in ComputeDistinguishablePairs. 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 ComputeDistinguishablePairs: 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 unreachable and redundant states. The task DFA contains u unreachable states and r redundant states. It is sensible to have $u > 1, r > 1$, such that RemoveUnreachables and RemoveEquivalentPairs will not be skipped. A exercise instructor will find it useful, to control exactly how big u and r are: The higher u, r, the more states have to be eliminated and merged.

Number of states alltogether ($|Q_{task}|$). The more states A_{task} has, the higher is the number of state pairs, which have to be checked. Thus a possiblity to adjust also the number of reachable and redundant states, denoted q, would be useful. Note that $|Q_{task}| = |Q_{inter}| + d = |Q_{sol}| + u + d$ so $q = |Q_{sol}|$.

Alphabet size ($|\Sigma|$). The more symbols the alphabet of A_{task}, A_{inter} and A_{sol} has, 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 ComputeUnreachables and RemoveUnreachables 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.

1.2.2 Definition and evaluation of possible requirements

Dismissed:

- $h(A_{sol}, A_{task}) = |Q_{task}| \ / \ |Q_{sol}|$
- number of transitions
- max degree of a node (Why not this?)
- Does GraphViz have a heuristic?

Accepted general criteria:

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

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

Accepted solution DFA criteria:

-> has to be minimal, complete

-> number of states

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

-> alphabet size

-> number of accepting states

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

-> completeness (follows)

-> A_{sol} is new

Definition 3 (New DFAs)**.** A DFA A_{sol} is *new* if it is not practically isomorph to any previously generated solution DFA.

Accepted intermediate 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 (can be checked in $O(|Q_{task}|)$)

-> 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}. As a consequence we need to ensure that adding redundant 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:

$q, a, f, m_{min}, m_{max} \in \mathbb{N}$,

$p \in \{0, 1\}$

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

A is new

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

Definition 5 (ExtendMinimalDFA)**.**

Given:

$A_{sol} = (Q, \Sigma, \delta, s, F) \in \mathcal{A}_{min}$,

$p \in \{0, 1\}$,

$d, u \in \mathbb{N}$

Request:

A DFA $A_{task} = (Q', \Sigma', \delta', s', F')$ with reachable redundant 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 $p = 1$,

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

The main algorithm will then simply be:

```
function GenerateDFAMinimizationProblem(q, a, f, m_min, m_max, p_1, p_2, d, u)
    A_sol ← BuildNewMinimalDFA(q, a, f, m_min, m_max, p_1)
    A_task ← ExtendMinimalDFA(A_sol, p_2, d, u)
    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 ComputeDistinguishablePairs 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 new

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.

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.

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} 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 ComputeDistinguishablePairs 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: $M' \leftarrow \{(p,q) \mid (p,q) \notin M \land \exists \sigma \in \Sigma\colon (\delta(p,\sigma), \delta(q,\sigma)) \in M\}$
7: $M \leftarrow M \cup M'$
8: **while** $M' \neq \emptyset$
9: $hasDupl \leftarrow |\{(p,q) \mid p \neq q \land (p,q) \notin M\}| > 0$
10: **return** $hasDupl, depth$

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

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

1: **function** HasUnreachableStates(A)
2: **return** $|$ComputeUnreachables$(A)| > 0$

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 [4] 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:

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

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:

function BuildNewMinimalDFA-2 $(q, a, f, m_{min}, m_{max}, p)$
    $l \leftarrow$ all DFAs in DB1 matching $q, a, f, m_{min}, m_{max}, p$
    while True **do**
        ...
        if A_{test} is practical isomorph to any DFA in l **then**
            continue
        save A_{test} and its respective properties in DB1
        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 $q_0, \ldots, q_{q-1}$. 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.

[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 $q \in Q$ 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.

function BuildNewMinimalDFA-3a $(q, a, f, m_{min}, m_{max}, p)$
    $l \leftarrow$ all DFAs in DB1 matching $q, a, f, m_{min}, m_{max}, p$
    $Q \leftarrow \{q_0, \ldots, q_{q-1}\}$
    $\Sigma \leftarrow \{\sigma_0, \ldots, \sigma_{a-1}\}$
    while True **do**
        $\delta \leftarrow \emptyset$
        for q **in** Q **do**
            for σ **in** Σ **do**
                $q' \leftarrow$ random chosen state from Q
                $\delta \leftarrow \delta \cup \{((q, \sigma), q')\}$
        $s \leftarrow 0$
        $F \leftarrow$ random sample of f states from Q
        $A_{test} \leftarrow (Q, \Sigma, \delta, s, F)$
        if A_{test} not minimal **or not** $m_{min} \leq \mathcal{D}(A_{test}) \leq m_{max}$ **then**
            continue
        if $p = 1$ **and** A_{test} is not planar **then**
            continue
        if A_{test} is isomorph to any DFA in l **then**
            continue
        save A_{test} and its respective properties in DB1
        return A_{test}

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 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 δ-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

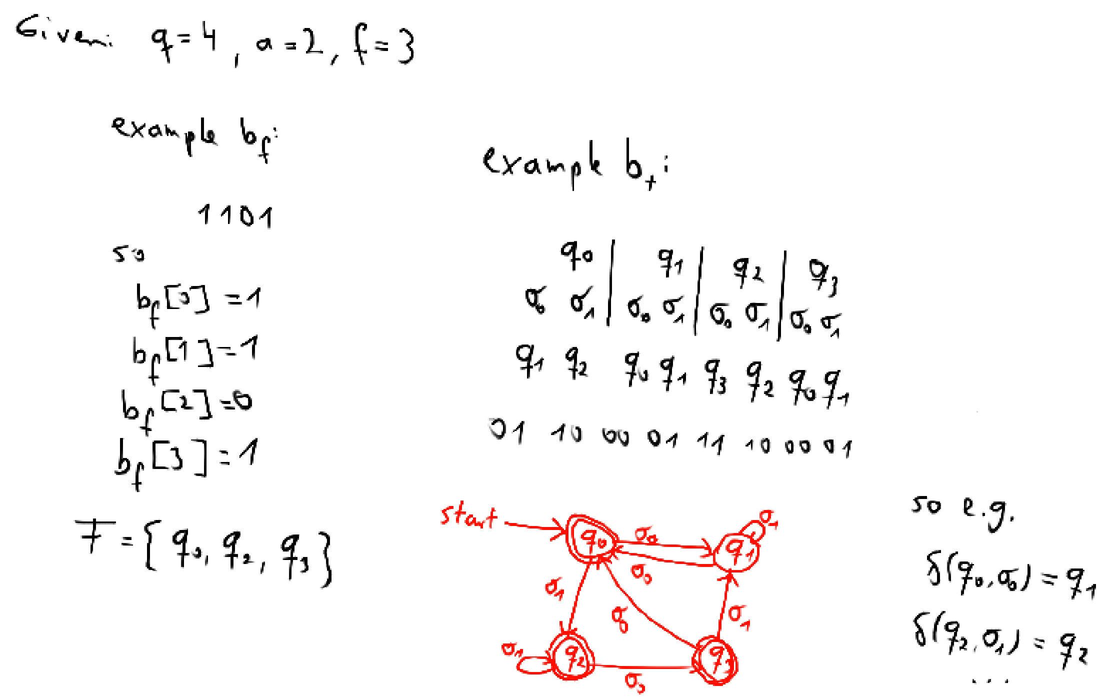

Figure 2.1: Example for two possible configurations of the bit-fields b_f and b_t given q, a and f. Below the corresponding DFA is drawn.

and b_t as $\log_2(q)$-ary. To get to the next DFA, we will first increment b_t by 1. If $b_t = 1 \ldots 1$, then we increment b_f until it contains f ones (again) and set b_t to $0 \ldots 0$. This behaviour is summarized in the following algorithm: **Gregor:** Clarify what happens at 11111...

function IncrementEnumProgress (b_f, b_t, q, a, f)
  add 1 to $(b_t)_2$
  if $b_t = 0 \ldots 0$ **then**
    while $\#_1(b_f) \neq f$ **do**
      add 1 to $(b_f)_2$
      if $b_f = 0 \ldots 0$ **then**
        return $\perp$
      $b_t = 0 \ldots 0$
  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:

function DFAfromEnumProgress (b_f, b_t, f)
  $Q \leftarrow \{q_0, \ldots, q_{q-1}\}$
  $\Sigma \leftarrow \{\sigma_0, \ldots, \sigma_{a-1}\}$
  $\delta \leftarrow \emptyset$
  for i **in** $[0, \ldots, q-1]$ **do**
    for j **in** $[0, \ldots, a-1]$ **do**
      $\delta \leftarrow \delta \cup \{((q_i, \sigma_j), q_{b_t[i*a+j]})\}$

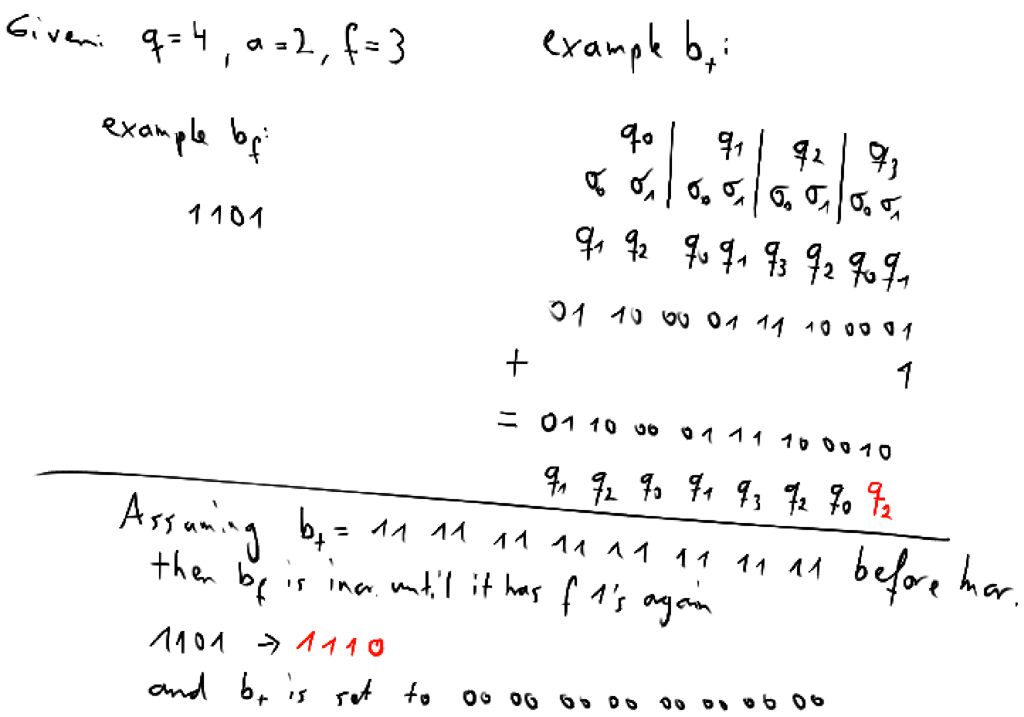

Figure 2.2: The upper half shows how a b_t-increment results in a change in the resulting DFAs transition function: $\delta(q_3, \sigma_1) = q_1$ becomes $\delta(q_3, \sigma_1) = q_2$. The lower half shows what happens, if b_t has reached its end.

$s \leftarrow q_0$
$F \leftarrow \{q_i | i \in [0, \ldots, q-1] \wedge b_f[i] = 1\}$
return $(Q, \Sigma, \delta, s, F)$

The initial bit-field values are each time $0 \ldots 0$. 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 Σ 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 to be not new. Thus we introduce a second database $DB2$ with the following table:

$$|Q_A| \quad |\Sigma_A| \quad b_f \quad b_t$$

We reduce the enumeration room for each calculation.

function BuildNewMinimalDFA-3b $(q, a, f, m_{min}, m_{max}, p)$
    $l \leftarrow$ all DFAs in DB1 matching $q, a, f, m_{min}, m_{max}, p$
    $b_f, b_t \leftarrow$ load enumeration progress for q, a, f, p from DB2
    while True **do**
        if b_f, b_t is finished **then**

save b_f, b_t
return $\perp$
$A_{test} \leftarrow$ next DFA based on b_f, b_t
if A_{test} not minimal **or not** $m_{min} \leq \mathcal{D}(A_{test}) \leq m_{max}$ **then**
continue
if $p = 1$ **and** A_{test} is not planar **then**
continue
if A_{test} is isomorph to any DFA in l **then**
continue
save b_f, b_t in DB2
save A_{test} and its respective properties in DB1
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-ComputeDistinguishablePairs iteratively. (Why would this basically result in running ComputeDistinguishablePairs 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 redundant states

-> number of unreachable states

-> alphabet size

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

-> completeness (for ComputeDistinguishablePairs-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 redundant 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 redundant states

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

Gregor: hidden definition: correct duplication

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

Consider the properties a redundant state, say q_d, must have. It is in particular equivalent 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 equivalent, iff $\forall \sigma \in \Sigma \colon [\delta(q_d, \sigma)]_{\sim_A} = [\delta(q_o, \sigma)]_{\sim_A}$. Thus, when adding some q_d, we have to choose for each symbol $\sigma \in \Sigma$ at least one transition from the following set:

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

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 by the equivalence 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 $[q_d]_{\sim_A} = [q_o]_{\sim_A}$ already. So, given a state s with $((s, \sigma), t)$ and $t \in [q_o]_{\sim_A}$, 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 redundant state.

So we have to choose at least one transition of

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

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:

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

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.

```
function AddRedundantStates (A_sol, d)
    eq_classes ← { {q} | q ∈ Q }
    eq_class(q) = C such that C ∈ eq_classes and q ∈ C
    #_{d^-}(q) = |d^-(q)|

    for d times do

        q_o ← ⊥
        for q in C do
            if #_{d^-}(q) ≥ 2 then
                q_0 ← random chosen state from eq_class(q)
                break
        if q_o = ⊥ then
            return ⊥

        q_d ← max Q + 1
        Q ← Q ∪ {q_d}
```

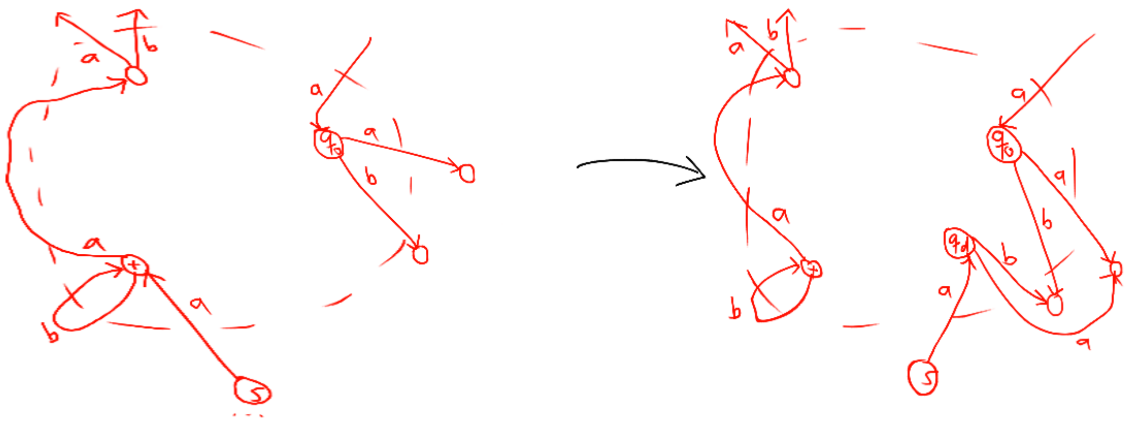

Figure 3.1: If an equivalence class (here denoted by the states in the dashed area) contains a state with 2 or more ingoing transitions (in this case t), then a state equivalent to any of the classes states may be added. Here q_d is equivalent to q_o and is “stealing” the ingoing transition $\delta(s,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

    for σ **in** Σ **do**
        $\delta(q_d, \sigma)$ = random chosen state from $eq\_class(\delta(q_o, \sigma))$

    $O \leftarrow \{ \ ((s,\sigma),t) \in \delta \mid t \in eq\_class(q_o) \wedge \#_{d^-}(t) \geq 2 \ \}$
    $C \leftarrow$ random sample of at least one transition from O
    for $((s,\sigma),t)$ **in** C **do**
        $\delta \leftarrow \delta \setminus \{((s,\sigma),t)\}$
        $\delta \leftarrow \delta \cup \{((s,\sigma),q_d)\}$
        $\#_{d^-}(t) \leftarrow \#_{d^-}(t) - 1$
        $\#_{d^-}(q_d) \leftarrow \#_{d^-}(q_d) + 1$
return A

3.1.1 Adding redundant states does not change L

p. 159 Hopcroft

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

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

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

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

Proof. See TI-Lecture ch. 4 “Minimization” p. 18. $\square$

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

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

Proof. Via direct proof.

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

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

m-ComputeDistinguishablePairs(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:

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

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 $p, q \in Q \colon (\delta^*(p, w) \in F \wedge \delta^*(q, w) \notin F)$.

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 3. *Adding redundant states to an automaton A does not increase the number of iterations in the* ComputeDistinguishablePairs*-algorithm for A.*

Proof. Proof per contradiction.

Let's assume adding redundant states $q_d^1, \ldots, q_d^n$ 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:

- $Q' = Q \cup \{q_d^1, \ldots, q_d^n\}$
- W.l.o.g. $\exists q_o^1 \in Q \colon \exists q_o^2 \ldots q_o^n \in Q \colon \quad \sim'_A (q_o^1, q_d^1), \ldots, \sim'_A (q_o^n, q_d^n)$

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

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

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:

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

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.

```
function AddUnreachableStates (A, u)
    for u times do
        q ← max Q + 1
        Q ← Q ∪ {q}
        R ← random chosen sample of |Σ| states from Q \ {q}
        for σ in Σ do
            q' ∈ R
            R ← R \ {q'}
            δ ← δ ∪ {((q, σ), q')}
    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 ComputeDistinguishablePairs-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.