Minimization of Acyclic DFAs

Johannes Bubenzer

University of Potsdam
Department of Linguistics
Karl-Liebknecht-Strasse 24-25
14476 Potsdam
bubenzer@uni-potsdam.de

Abstract. There exists a linear time algorithm for the minimization of acyclic deterministic finite-state automata (ADFA) due to Dominique Revuz [17]. The algorithm has different phases involving the computation of a *height* for each state, sorting states of the same height and joining equivalent states. We present a new linear time algorithm for the task at hand. The algorithm is conceptionally simpler and computes the minimal automaton in only a single depth-first traversal over the automaton. The algorithm utilizes the notion of a right-language *Register* known from algorithms for the incremental construction of minimal ADFAs from lists. In an evaluation we compare the running times of both algorithms. The results show that the new algorithm is significantly faster than Revuz' algorithm.

Keywords: minimization, acyclic deterministic finite state automata, algorithmic

1 Introduction

The DFA minimization problem dates back to the 1950s with the works of [12] and [14]. In the meantime a multitude of different minimization algorithms were discussed. There are several different known algorithms, with different running times and properties. Additionally, there are algorithms that only minimize automata of specific types. In this paper we present a new algorithm for the minimization of acyclic DFAs and compare it to Revuz' well-established algorithm [17] for this problem.

In section 2 we give an overview on different minimization algorithms for the general case of DFAs. Mathematical preliminaries are presented in section 3. Then we describe the best known minimization algorithm for acyclic DFAs and its proof in section 4. In section 5 we propose a new minimization algorithm for the acyclic case. We prove its correctness and discuss its advantages over the former algorithm. Finally, we describe the results of an evaluation comparing both algorithm (section 6).

2 Overview

First, we describe DFA minimization algorithms that do not pose any restriction about acyclicity. The now classical Moore-Algorithm [14] shows $O(n^2)$ time and memory complexity for n being the number of states of the automaton to be minimized. It works by marking non-equivalent states. This procedure starts with pairs of final and non-final states marked as non-equivalent. In the subsequent steps all those states leading into non-equivalent states are marked as non-equivalent, until no more states to be marked are found. The fastest known algorithm is the Hopcroft algorithm [10], it runs in $O(n \log n)$. Due to the complexity of the algorithm and its proof, a few papers were published that explain and re-explain the Hopcroft algorithm ([8], [13]).

Johannes Bubenzer: Minimization of Acyclic DFAs, pp. 132–146.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2

The algorithm is derived from the classical Moore-Algorithm and maintains sets of possibly equivalent states. Those state-sets are then split up in a special way such that the number of splitting operations along with the costs of splitting a state-set leads to the aforementioned complexity.

There is also an incremental algorithm [21], further enhanced in [23]. The algorithm determines the minimal automaton in cubic time, but can be halted at any time – resulting in a partially minimized automaton.

The Brzozowski-algorithm [2] poses a notable exception to the algorithms described here. The algorithm determines the minimal automaton $\mathcal{A}_m$ by applying a chain of regular operations to the source automaton $\mathcal{A}$.

$$\mathcal{A}_m \longleftarrow det(inv(det(inv(\mathcal{A})))) \tag{1}$$

The inner automaton inversion $inv()$ and determinization $det()$ are minimizing the suffix part of the automaton. The outer inversion and determinization then minimize the prefix part. The determinization operation used is the subset method, which ensures that the suffix part is only broken up if required for the prefix compression. The algorithm shows exponential worst case complexity, since the complexity of determinization is exponential. But it performs exceptionally well in practice [4].

For certain subsets of the deterministic finite-state automata, one can achieve minimization in linear time. The best know algorithm is due to Revuz [17] – it minimizes acyclic DFAs in linear time. This approach is discussed in depth in section 4. [1] extend Revuz' algorithm and show that the O(n)-complexity bound can also be achieved for a bigger subclass, the so called single-cycle automata. These are automata where each state can have at most one outgoing transition leading into a cycle.

There also exist algorithms for the direct compilation of minimal ayclic DFAs from lists of words. [22] poses an excellent and profound summary of the work in this field. Furthermore, [3] describes improved algorithms for the task at hand.

A taxonomy of the most important finite state minimization algorithms can be found in [19] and in [20]. The presented paper is an elaboration on parts of the work done in the author's thesis [3].

3 Preliminaries

In this section we introduce the basic definitions and theorems regarding languages and automata needed in this paper. A more complete introduction using similar notions can be found in [11].

3.1 Alphabets, Words and Languages

An *alphabet* Σ is a finite set of symbols. Any finite sequence of concatenations of symbols $w = a_1 \cdot a_2 \cdots a_n \mid a_i \in \Sigma$ from Σ is called a *word* over Σ. The symbol $w[i] = a_i \in \Sigma$ of a word w is the symbol at the i'th position. The length of a word $|w|$ is the number of concatenated symbols. By ε we denote the empty word ($|\varepsilon| = 0$). We denote by Σ^* the *free monoid* generated by Σ. That is the set of all words over Σ including ε, along with identity ε and the concatenation $\cdot$ as operation. By Σ^+ we denote the *free semigroup* generated by Σ, that is the set of all non-empty words together with the concatenation operation. Any subset $L \subseteq \Sigma^*$ of Σ^* is called a *language*. For convenience we assume each alphabet to be a total order together with some operation $<$.

3.2 Automata and Languages

A *deterministic finite-state automaton* (DFA) $\mathcal{A} = \langle Q, q_0, \Sigma, \delta, F \rangle$ consists of a finite set of states Q, a designated start-state $q_0 \in Q$, an alphabet Σ, a set of final states $F \subseteq Q$ and a transition function $\delta : Q \times \Sigma \mapsto Q$. The transition function is extended to the acceptance of words in the usual way:

$$\delta^*(q, \varepsilon) = q$$
$$\delta^*(q, aw) = \delta^*(\delta(q, a), w)$$

for all states $q \in Q$ and $a \in \Sigma$ is a symbol, $w \in \Sigma^*$ is a word. A word w is said to be *accepted* by the automaton, if $\delta^*(q_0, w) \in F$. The language $\mathcal{L}(\mathcal{A})$ accepted by a DFA $\mathcal{A}$ is the set of words accepted by $\mathcal{A}$:

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$$

Each language accepted by some DFAs is called a *regular language*.

The *right-language* $\overrightarrow{\mathcal{L}}(q)$ of a state $q \in Q$ is defined as the set of words accepted by an automaton started in q:

$$\overrightarrow{\mathcal{L}}(q) = \{w \in \Sigma^* \mid \delta^*(q, w) \in F\}$$

We denote by $\delta'(q)$ the set of outgoing transitions from state q:

$$\delta'(q) = \{\langle a, p \rangle \mid a \in \Sigma,\ p \in Q,\ \delta(q, a) = p\}$$

The destination state of a transition $t \in \delta'(q)$ is denoted by t_{next} and the transition symbol by t_{sym}.

A typical way to implement the set of outgoing transitions is as an array of transitions. We use the term *transition array* in the following to refer to the actual implementation of the set of outgoing transitions.

We define the *signature* $\overrightarrow{q}$ of a state q, to be q's set of outgoing transitions unified with ε iff the state is final:

$$\overrightarrow{q} = \begin{cases} \delta'(q) \cup \{\varepsilon\} & \texttt{if } q \in F \\ \delta'(q) & \texttt{else} \end{cases}$$

In the latter we will often refer to the signature as being a sequence of finality attribute and states rather than a set. A sequential signature holds the transitions ordered by the transition symbol, the finality attribute being the first element if present.

A state q is called *accessible* if it is reachable from the start-state:

$$\exists w \in \Sigma^* \mid \delta^*(q_0, w) = q$$

A state q is said to be *co-accessible* if there is a path from q to a final state.

$$\exists w \in \Sigma^* \mid \delta^*(q, w) \in F$$

A DFA is *connected* iff each of its states is both accessible and co-accessible. Each non-connected DFA can be easily transformed into a connected DFA by removing all non-accessible and all non-co-accessible states as well as transitions from and

into these states from the automaton. Note that this operation does not change the language of the automaton.

A DFA is called *acyclic* deterministic finite-state automaton (ADFA) if no state is connected to itself by a chain of transitions and *cyclic* otherwise.

We will assume in the following, that the transition arrays of states are sorted by their transition symbols. We can assume this without loss of generality as shown in the following proof.

Lemma 1. *We can sort the transition arrays of all states of a DFA in linear time.*

Proof. We are looking at DFAs and therefore the number of outgoing transitions from one state is bounded by the constant $|\Sigma|$. Since Σ is a constant, we can sort the transition array of a single state in time relative to $|\Sigma|$. This is done for each state in $|Q|$ once. Therefore the computation takes linear time wrt. the size of the automaton.

3.3 Minimality

One important property of DFAs is that they have a state *minimal* representation. This ensures small automata for given languages. It follows from the Myhill-Nerode theorem ([15], [16]), that for each regular language $\mathcal{R}$, there exists exactly one *minimal deterministic finite-state automaton* (MDFA) $\mathcal{A}$ (excluding isomorphism) accepting the language $\mathcal{R}$ ($\mathcal{L}(\mathcal{A}) = \mathcal{R}$) with a minimal number of states. Further it follows that the minimal automaton for a given language $\mathcal{L}$ is exactly that automaton accepting $\mathcal{L}$ which has only states with mutually different right-languages:

Definition 2 (Minimal DFA). *A connected DFA $\mathcal{A}$ is minimal if all states have mutually different right languages:*

$$min(\mathcal{A}) \iff \forall q, p \in Q : \overrightarrow{\mathcal{L}}(q) \neq \overrightarrow{\mathcal{L}}(p) \tag{2}$$

We call two states $p, q \in Q$ equivalent iff they have the same right-languages ($\overrightarrow{\mathcal{L}}(q) = \overrightarrow{\mathcal{L}}(p)$) and inequivalent otherwise.

Theorem 3 (Minimal ADFA). *An acyclic DFA is minimal if all states have mutually different right-language signatures.*

Proof. Let us assume that in an acyclic connected DFA $\mathcal{A}$ all states have different signatures but at least two states accept the same right-language: Then the following statement must be true:

$$\exists p, q \in Q : \overrightarrow{\mathcal{L}}(q) = \overrightarrow{\mathcal{L}}(p) \wedge \forall q, p \in Q : \overrightarrow{q} \neq \overrightarrow{p} \tag{3}$$

$$\Rightarrow \exists p, q \in Q : \overrightarrow{\mathcal{L}}(q) = \overrightarrow{\mathcal{L}}(p) \wedge \overrightarrow{q} \neq \overrightarrow{p} \tag{4}$$

$$\Leftrightarrow \exists p, q \in Q : \overrightarrow{\mathcal{L}}(q) = \overrightarrow{\mathcal{L}}(p) \wedge (q \in F \neq p \in F \tag{5}$$

$$\vee \exists a \in \Sigma : \delta(q, a) \neq \delta(p, a)) \tag{6}$$

It is impossible that both states have equal right languages if one is final and the other is not. Therefore both states have to differ in their outgoing transitions to fulfill the last equation. This could be for two reasons.
(1) There is a transition for a symbol $a \in \Sigma$ for one state but not for the other. This would violate the equation since then one state accepts a word starting with a while the other doesn't and their right-languages would differ. (2) There is a transition with

symbol $a \in \Sigma$ leading into a state q_2 for q and to p_2 for p and $q_2 \neq p_2$. By precondition we know that both q_2 and p_2 have different right-language signatures. Further they need to have the same right-languages to make the statement become true. But then by the same argument there must exist two states q_3 and p_3 and so on up to infinity. This is impossible since the automaton is acyclic.

Theorem 4 (Equal signatures). *Two states $q, p \in Q$ with equal right language signatures ($\overrightarrow{q} = \overrightarrow{p}$) are equivalent.*

Proof. Since both states lead into the same destination states with the same labels and are either both final or non-final they are equivalent, obviously.

Definition 5 (Minimal states). *A state $q \in Q$ is called a minimal state iff no equivalent state p exists in the underlaying automaton and all successor states of q are minimal. A final state with no successor states is a minimal state iff no other final state without successor states exists.*

Definition 6 (Minimal signatures). *The signature of a state $q \in Q$ is called minimal signature $\overrightarrow{q}_m$ iff all of its successor states are minimal states.*

Theorem 7 (Differing minimal signatures). *Two states $q, p \in Q$ of an ADFA with different minimal signatures ($\overrightarrow{q}_m \neq \overrightarrow{p}_m$) are not equivalent.*

Proof. Let us assume that the opposite would be true and that two equivalent states $q, p \in Q$ with different minimal signatures exist. Then we arrive at equation 2 of the proof of theorem 3. We already showed there that such a situation is impossible in an acyclic DFA.

4 Algorithm for the acyclic case

The case of minimization of acyclic DFAs can be done in linear time, wrt. to the size of the input automaton. In the following we describe the linear time algorithm for acyclic connected DFA by [17]. This algorithm consists of two phases. In the first phase a height is calculated for each state and all states are grouped together according to the their height. The *height* of a state is its maximal distance to a final state. In the second phase the states of the same height are sorted according to their signature starting with states of the lowest height and proceeding to the highest. States of the same height and signature are equivalent according to their right-language (given that all states of smaller height are already minimized). Those equivalent states are then minimized and incoming transitions are adjusted accordingly. The critical computation is the sorting of the states. As shown below this can be done in linear time, using a special kind of bucket-sort. In the following we will informally describe Revuz' algorithm.

Definition 8. *The height h_q of a state q is the length of the longest path to a final state. And can be defined with the following recursive formula.*

$$h_q = \begin{cases} 0 & if\ \delta'(q) = \emptyset \\ 1 + max(h_p) : \langle a, p \rangle \in \delta'(q) & else \end{cases}$$

Theorem 9 (Computing heights). *The computation of the heights of all states of a DFA $\mathcal{A}$ can be done in linear time.*

Proof. We show this by proving that the height can be computed by a depth first traversal. In a depth first traversal over an acyclic DFA, it is guaranteed that for each state q all successor states are processed before q is fully processed. Therefore a depth first traversal over $\mathcal{A}$ is conducted. If a state q has no outgoing transitions, it is of height 0. Otherwise the heights of its successors are computed and q receives a height of one plus the height of the successors with the biggest height as indicated in definition 8. A depth first traversal takes linear time.

After the height of a state q is computed, q is inserted into a partition P_{h_q} of states with height h_q. Partitions are stored in an array of size $|Q|$, since the maximum possible height of a state is $|Q| - 1$ if all states form a single path. In the next step, states of the same height are sorted according to their signature and all states with equal signature are then merged into one.

Radix Sort Radix sort [18], a variant of bucket sort, is a sorting algorithm which can sort an array of (sequential) elements in linear time. The elements to be sorted are sequences of singular values, drawn from some fixed alphabet (for example bytes). The alphabet has to be known beforehand. Initially, buckets are created, one for each symbol in the alphabet. The elements are sorted into those buckets according to their least significant value. Then, again the elements in the buckets are rearranged according to the next least significant bit and so on. When different length sequences occur, the shorter sequences are treated as if their most significant values were padded with the lowest alphabet symbol. In each pass, at most n elements are rearranged and the rearrangement happens at most as often as the length of the longest sequence. Therefore the algorithm takes linear time.

Sorting states of the same height Given an array of states of the same height, one can use radix sort to sort the array in linear time. In this case the size of the alphabet $|\Sigma|$ appears as constant factor k in the complexity bound. To improve this, Revuz suggests the following computation. An array A of size $|\Sigma|$, which is preinitialized with zero is held. This is used to temporary rewrite the signatures of all states of the current height level. Now, all elements of the signatures of the states are scanned and each element is looked up in the array A. If the element is already present, the signature is rewritten by the value of the element in the array. Otherwise, the element is inserted into the array and a value of one plus the number of elements currently in the array is assigned to it. Additionally, the element is pushed onto a pushdown store used after the sorting to reinitialize the array A. After rewriting all signatures, radix sorting can be done with a constant factor k equal to the number of distinct elements in the signatures of the states of the current level.

A somewhat simpler way with the same complexity would be to use a hash to index the states of one height. Equivalent states would then be merged to the same slot.

Theorem 10 (Linear time). *Minimization using Revuz algorithm takes linear time.*

Proof. Computation of the height partitioning for all states takes linear time. Sorting the states of one height partition takes linear time, wrt. the size of the partition. Determining states of equal signature obviously takes linear time, too, since equivalent states are sorted to adjacent positions. Rewriting equivalent states (and the transitions of their successors) such that one class representative is chosen, is also a linear

time operation. Since each height partition is computed exactly once and the number of states of all partitions sum up to $|Q|$. We conclude that minimization according to Revuz takes linear time.

As was just shown, all the individual steps performed by the algorithm take linear time. It remains to show that the algorithm indeed computes the minimal automaton. We will show this with the following recursive proof. First, we show that upon the computation of a certain height level, all states of that level have minimal signatures.

Theorem 11 (Minimal signatures). *States of a certain height level have minimal signatures when they are considered.*

Proof. The algorithm starts by considering states of height 0 states that are final and have no outgoing transitions. Those states have minimal signatures, obviously.
Given a state q of height $h_q > 0$ is considered. Since in that case all successor states of q have smaller heights, they were considered before and are already minimized by assumption. Therefore the state q has a minimal signature.

As shown in theorem 4, states with equal signatures are equivalent and can therefore be merged together. Next, we show that upon merging all equivalent states of the given height wrt. their signatures, all states of that height are truly minimal.

Theorem 12 (After computation of a height level all of its states are minimal).

Proof. A state is minimal iff no other state with the same right language exists. We show by contradiction that this is impossible. Let us assume there exists a equivalent state p of height h_p (with the same right language) as a state q of the current height h_q after merging the states of height level h_q. The state p could be
(1) *of smaller height*. Since the longest path from q to a final state has length h_q and the longest path of p has length h_p and $h_q > h_p$, the right language of q contains a word of length h_q which is not in the right language of p. Then q and p can not be equivalent.
(2) *of bigger height*. Since the longest path from q to a final state has length h_q and the longest path of p has length h_p and $h_p > h_q$, the right language of p contains a word of length h_p which is not in the right language of q. Again, both states can not be equivalent then.
(3) *of the same height*. Then p and q have different signatures since otherwise they would have been merged together. But the signatures are minimal as shown above. By theorem 7 we known that the states can not be equivalent.

Theorem 13 (Minimal ADFA). *The algorithm computes the minimal ADFA.*

Proof. The algorithm computes all height levels up to the biggest. By theorem 12 we known that upon the computation of a height level all of its states are minimal. Also, no states of lower heights are changed within the computation of a specific height level. Therefore after computation of the biggest height level all states in the automaton are minimal.

5 Improved algorithm for the acyclic case

In the following we describe a new algorithm for the minimization of acyclic connected DFAs. We prove its correctness and linear running time. The algorithm turns out to be faster in practice than Revuz' as supported by experimental evaluation in section 6. It is also more intuitive in our opinion. Daciuk [5] has described a similar algorithm for minimizing tries. But surprisingly the generalization to arbitrary ADFAs was not described in the literature before.

The algorithm maintains a data-structure mapping right-languages to states, which we will call the *Register*. This terminology was introduced by [6] in the context of compiling lists of words to MDFAs. In the following, we will treat the Register as if it stores right-languages as keys. Since the right-language of a state can contain up to $|\Sigma|$ transitions and there are up to $|Q|$ distinct right-languages for each DFA, such a Register would require $O(|\Sigma|n) = O(n)$ space at the most. Using a hash-table, lookup and storage of a right-language requires constant time. Nevertheless, in practical applications one could compute an integer hash-key from the right-language transition-array using a sufficient hash-function. The hash value does contain a pointer to the desired state then. Doubly representation of the transition array (in the state and as hash key) can therefore be avoided. Then the constant $|\Sigma|$-factor would disappear in the memory-complexity estimation of the Register.

A mapping from states to minimized states (*StateMap*) is filled by the algorithm. The *StateMap* is required since we need to be able to detect if some state encountered by the algorithm were already minimized before.

The algorithm merely consists of a depth-first traversal over the states of the automaton. If it encounters a state q with no outgoing transitions, q has the trivial right-language signature $\overrightarrow{q} = \langle\varepsilon\rangle$. Note that such a state is always final, because we are assuming connected DFAs. If no state with this right-language was encountered before, q is the new representative of the class and is kept. Otherwise q is replaced by the trivial state encountered first. If the algorithm encounters a state q with at least one outgoing transition, the decision on the minimal right-language of q is delayed until all successor states are minimized. Then q will also either form the representative of a new class of minimized states, or be replaced by the earlier detected representative of q's right-language class. The (recursive) pseudo-code is given in algorithm 1. For convenience, we assume the algorithm to be a method of the ADFA to be minimized. To minimize an ADFA the method is called with the start state q_0, an empty *Register* and the *StateMap* as arguments.

In the following, we will prove that the algorithm indeed computes the MDFA for a given DFA. As precondition we require the input automaton to be deterministic, acyclic and connected.

Definition 14. *We define a strict partial order $<$ over the states of an (acyclic) DFA, such that*

$$\forall p, q \in Q : p < q \iff \exists \alpha \in \Sigma^+ : \delta^*(p, \alpha) = q \tag{7}$$

Lemma 15. *The algorithm computes the states wrt. the order given by $<$.*

Proof. The algorithm performs a depth-first (preorder) traversal. This kind of traversal is known to compute states wrt. the order given by $<$.

Algorithm 1: Depth-first minimization of acyclic DFAs

```
1  begin minimize(State q, Register R, StateMap M)
2      foreach Transition t ∈ δ'(q) do
3          if M[t_next] == undef then
4              minimize(t_next, R, M)
5          t_next = M[t_next]
6      if R[q→] == undef then
7          M[q] = R[q→] = q
8      else
9          M[q] = R[q→]
10         deleteState(q)
```

We will prove that the algorithm indeed computes the right MDFA given a DFA in the following way. We state as a loop invariant, that at the beginning of the execution of the algorithm no states are mapped to wrong representatives (of their right language), and therefore no errors were made so far. We will then show by contradiction, that a situation where the loop invariant changes (that is: an error is introduced) can not exist.

Lemma 16. *The DFA is correctly minimized after the execution of the algorithm.*

Proof. Assume Lemma 16 does not hold. This means that either two states with the same right language exist or the FSA is not deterministic or not connected anymore.

Deterministic: The input automaton is deterministic by precondition. This could only change if either transitions are altered or added. Transitions are only altered in line 5, but there only the destination state changes. No transitions are added. We conclude that the automaton is deterministic afterwards.

Connected: The input automaton is connected by precondition. To lose this property requires either the finality attribute of a state to change or a state to be added or a state to be deleted leaving its incoming transitions or transitions to be deleted leaving their destination states unreachable. The finality attribute never changes. No states are added. States are only deleted in line 10. But each state that is deleted is marked in the $StateMap$ by an equivalent state. Deletion occurs only at the first invocation of a given state q. At each later invocation of q, the equivalent state is used (in line 5) and all occurrences (as destination of transitions) of q are replaced by it. We conclude that the automaton is connected afterwards.

Equivalent states: Let us assume there are at least two distinct states with the same right-language after minimization. The states in $StateMap$ are those which are already treated and minimized. The assumption implies that after the execution of the algorithm, there are at least two states in $StateMap$ that are mapped to distinct states, but are equivalent nevertheless. At the beginning of the algorithm there is no state registered in $StateMap$. Therefore the invariant holds, that at that time no two states are minimized and mapped to distinct states, but are equivalent. That means it must be possible to identify a line in the algorithm where the invariant changes. That is, before the execution of the line invariant holds and afterwards it doesn't hold anymore. This could only happen in line 7 or 9, since only there changes to the $StateMap$ are made.

Let us assume the invariant changes in line 7, this means we are treating a state q, which has a right language that was not seen before. Since we have not seen the q's right-language before (but by invariant all right-languages were computed correctly so far), no state can possibly be equivalent to the current one. Therefore the only reason for an error at this point could be, that q's right-language was computed wrong. We know from the strict partial order $<$ that all successor states were visited and minimized before the current state. We also know that they are correct and that the right-language vector is sorted. Since the right language results from the direct successor states q, the only possibility that q's right-language is wrong is, that the computation of the right-language of successor state's were wrong. This contradicts the assumption, that the invariant changes at this point and can therefore not be true. Also if q's right-language is trivial (q has no successors) no error could possibly be introduced.

Given the invariant changes in line 9. There, a state p with the same right-language as the state q is found and the state q is mapped onto p. To fulfill the assumption that an error is introduced, this mapping has to be wrong. This means that the class that q is mapped to is not the class q belongs in. p's right-language cannot be wrong by invariant. Therefore the right-language assigned to q must be wrong. This cannot be true, by the same argument as in the case of line 7.

Lemma 17. *The minimization takes linear time in the number of states of the input automaton.*

Proof. The algorithm performs a depth-first traversal. Each state is processed exactly once in the function *minimize*. Also the destination state of each transition of each state is processed and redirected at most once. Lookup and storage in the state register and state map require constant time. Therefore the minimization takes linear time.

5.1 Comparison

In comparison to Revuz' algorithm the new one neither requires an external sorting phase, nor does it require states to be sorted into buckets. All work is done in the single depth-first traversal. This makes the new algorithm quite easy to understand and implement. The complexity of Revuz' algorithm is hidden in the sorting phase, whereas most of the complexity of the new algorithm lays in the implementation of the Register. The register can be implemented as a general purpose hash, whereas Revuz' sorting is not general purpose. We believe that the new algorithm is therefore the better choice for the minimization of DFAs. Since Revuz' also requires factorization of states into buckets, we also believe that the constant factors of the new algorithm are lower and that it will execute faster. In the evaluation section we will strengthen this statement by showing that the new algorithm is faster in practice.

6 Evaluation

The new algorithm achieves improvements regarding the constant factors of the running time over the original algorithms but the asymptotic time-complexity is unchanged. To asses the advantage of the new algorithm we conducted experiments on random data as well as on natural language data sets. In this section we present the

results of our evaluation. As input to the minimization algorithms we used trie-DFAs compiled from randomly generated word lists. In the following we describe the random lists along with the sampling methods used to generate the data sets. Thereafter we describe the natural language data-sets used in the evaluation. Finally we present the results for the new minimization algorithm in comparison to the original algorithm. The experiments in this section were conducted on a computer with 4-core Intel Core^{TM} i5 750 processor (2.66 GHz), 8 GB of RAM running a Linux operating system with 64-bit architecture.

6.1 Random Data

We tried to design our experiments such that the influence of different parameters can be assessed from the results. The following parameters were varied in the individual experiments:

1. alphabet size
2. maximum string length
3. length of the input list
4. sampling distributions

We performed of our experiments on random data sets using two different alphabet sizes, namely $|\Sigma| = 5$ and $|\Sigma| = 50$. For the maximum string length we evaluated lists of short strings of at most 10 characters and lists of long strings of up to 50 characters. We varied the length of the input lists over the values $\{1000, 100000, 200000, \ldots, 1000000\}$ for each experiment. We used two different random sampling distributions to generate the words of the random lists. On the one hand the uniform (discrete) random distribution over strings up to the maximum length were evaluated. On the other hand a random distribution where each string-length up to the maximum string length gets the same probability whilst strings of a certain length are uniformly distributed.

In the uniform random distribution each string w over the alphabet Σ of length up to the maximum length l has the same probability $p(w)$:

$$p(w) = \frac{1}{\sum_{i=0}^{l} |\Sigma|^i} \tag{8}$$

The uniform distribution produces far more strings of the maximum length than of smaller lengths.

Under the distribution with equally distributed lengths a word w with length $|w| = n$ over the alphabet Σ with maximum word length l has the probability $p(w)$:

$$p(w) = \frac{1}{l|\Sigma|^n} \tag{9}$$

Note that we excluded the empty word ε from being generated by this second distribution.

For each combination of the different parameters described we created 16 different random lists. Running times were obtained by running both algorithms on a trie obtained from each random list. The average of the different runs on a specific parameter combination were computed for each algorithm. We actually report the percentage rates of the running times of the different algorithms. For each experiment the running time of the slowest algorithm were considered as 100 %.

Table 1. Performance of ADFA minimization methods under uniform distribution. String lengths and alphabet sizes vary over the plots.

We implemented the new minimization algorithm in and integrated it into the FSM⟨2.0⟩-library [9] in the C++ programming language. FSM⟨2.0⟩ contains methods to create, maintain, save and load automata which enabled us to build up the DFAs required for the experiments, easily. The library also contains a well optimized version of Revuz' algorithm and was therefore suited for (unbiased) experiments comparing both minimization algorithms. All experiments involved two phases. The performance of the second phase was actually measured.

1. In the first phase a trie-DFA was build up from a list of words. This DFA was then stored in a binary file (which can be loaded fast).

Table 2. Performance of ADFA minimization methods on data with uniform distributed lengths. String lengths and alphabet sizes vary over the plots.

In table 1 the results of both minimization algorithms are compared for data sampled from the uniform distribution. It is evident that the new algorithm performs a lot faster than the original one. The gap between both algorithm grows both with increasing alphabet size and increasing maximum word length. The difference for short lists is quite small. This is because the program overhead of initialization, preallocation and loading the unminimized automaton is quite big in comparison to the actual minimization algorithm. The curve for the new algorithm shows a slope for very large lists in the plot with alphabet size $|\Sigma| = 50$ and maximum string length 50. The reason may be that the implementation of Revuz' algorithm uses too much memory and begins to swap data to the hard drive. To make a profound statement about this further research is required.

The algorithms perform comparable on the lists generated with uniform distributed lengths (table 2). The differences between both implementations are smaller.

Language	Revuz	Bubenzer
German	1.0	0.76
Rnglish	1.0	0.73
French	1.0	0.73
Dutch	1.0	0.76

Table 3. Performance of ADFA minimization algorithms on natural language data sets.

This is because the total amount of data is smaller in this case since the distribution produces shorter words far more frequently. For the same reason the slope that occurred in the last experiment doesn't arise in the plots.

We obtained similar results for the natural language data-sets as can be seen in table 3. The word lists are quite small in comparison to the random lists. It can be expected that the gap between both algorithms would increase with longer input lists in a similar way as with the random data.

7 Conclusions

We presented a new minimization algorithm for acyclic DFAs and proved its correctness. Further we evaluated the performance of the algorithm against Revuz' well-established algorithm for this case. Our results show that the new algorithm is significantly faster in practice.

References

1. J. Almeida and M. Zeitoun: *Description and analysis of a bottom-up DFA minimization algorithm.* Information Processing Letters, 107(2) 2008, pp. 52–59.
2. J. A. Brzozowski: *Canonical regular expressions and minimal state graphs for definite events*, in Mathematical theory of Automata, Volume 12 of MRI Symposia Series, Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y., 1962, pp. 529–561.
3. J. Bubenzer: *Construction of minimal ADFAs*, Diplomarbeit, Universität Potsdam, Germany, 2011.
4. J.-M. Champarnaud, A. Khorsi, and T. Paranthoën: *Split and join for minimizing: Brzozowski's algorithm*, in PSC'02: The Prague Stringology Conference '02, Czech Technical University in Prague, 2002.
5. J. Daciuk: *Comparison of construction algorithms for minimal, acyclic, deterministic, finite-state automata from sets of strings*, in Implementation and Application of Automata, Springer, 2003, pp. 127–152.
6. J. Daciuk, B. W. Watson, S. Mihov, and R. E. Watson: *Incremental construction of minimal acyclic finite-state automata.* Computational Linguistics, 26(1) 2000, pp. 3–16.
7. Deutscher Wortschatz: `http://wortschatz.uni-leipzig.de`, 2011.
8. D. Gries: *Describing an algorithm by Hopcroft.* Acta Informatica, 2 1973, pp. 97–109.
9. T. Hanneforth: *fsm2 – A scripting language interpreter for manipulating weighted finite-state automata*, in Anssi Yli-Jyrä et al. (eds): Finite-State Methods and Natural Language Processing, 8th International Workshop, Berlin, 2009, Springer, pp. 13–30.
10. J. E. Hopcroft: *An $n \log n$ algorithm for minimizing the states in a finite automaton*, in The Theory of Machines and Computations, Z. Kohavi, ed., Academic Press, 1971, pp. 189–196.
11. J. E. Hopcroft and J. D. Ullman: *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Series in Computer Science, Addison-Wesley Publishing Company, Reading, MA, 1979.
12. D. A. Huffman: *The synthesis of sequential switching circuits.* Journal of the Franklin Institute, 257(3) 1954, pp. 161–190.

13. T. Knuutila: *Re-describing an algorithm by Hopcroft.* Theor. Comput. Sci., 250 January 2001, pp. 333–363.
14. E. F. Moore: *Gedanken-experiments on sequential machines*, in Automata Studies, Princeton, 1956, pp. 129–153.
15. J. Myhill: *Finite automata and the representation of events*, Tech. Rep. WADD TR-57-624, Wright Patterson Air Force Base, Ohio, 1957.
16. A. Nerode: *Linear automaton transformations.* Proceedings of the American Mathematical Society, 9 1958, pp. 541–544.
17. D. Revuz: *Minimization of acyclic deterministic automata in linear time.* Theoretical Computer Science, 92(1) 1992, pp. 181–189.
18. R. Sedgewick: *Algorithms in C++, 3rd Ed.*, Addison-Wesley, Reading, MA, 1998.
19. B. W. Watson: *A taxonomy of finite automata construction algorithms*, Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, 1993.
20. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Eindhoven University of Technology, the Netherlands, 1995.
21. B. W. Watson: *An incremental DFA minimization algorithm*, in ESSLLI Workshop 2001, 2001.
22. B. W. Watson: *Constructing minimal acyclic deterministic finite automata*, PhD thesis, University of Pretoria, Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa, 2010.
23. B. W. Watson and J. Daciuk: *An efficient incremental DFA minimization algorithm.* Nat. Lang. Eng., 9 March 2003, pp. 49–64.