Minimization of automata

Jean Berstel[1], *Luc Boasson*[2], *Olivier Carton*[2], *Isabelle Fagnot*[1,*]

[1]Laboratoire d'Informatique Gaspard-Monge
Université Paris-Est Marne-la-Vallée
5, boulevard Descartes, Champs-sur-Marne, F-77454 Marne-la-Vallée Cedex 2

[2]LIAFA
[*]Université Paris Diderot
Case 7014, F-75205 Paris Cedex 13

email: `{berstel,fagnot}@univ-mlv.fr`, `{boasson,carton}@liafa.jussieu.fr`

January 4, 2011 3 h 39

2010 Mathematics Subject Classification: 68Q45

Key words: Finite automata, minimization, Hopcroft's algorithm.

Contents

1 Introduction

This chapter is concerned with the design and analysis of algorithms for minimizing finite automata. Getting a minimal automaton is a fundamental issue in the use and implementation of finite automata tools in frameworks like text processing, image analysis, linguistic computer science, and many other applications.

There are two main families of minimization algorithms. The first by a sequence of refinements of a partition of the set of states, the second by a sequence of fusions or merges of states. Among the algorithms of the first family, we mention a simple algorithm described in the book [35]. It operates by a traversal of the product of the automaton with itself, and therefore is in time and space complexity $O(n^2)$. Other algorithms are Hopcroft's and Moore's algorithms, which will be considered in depth later. The linear-time minimization of acyclic automata of Revuz belongs to the second family. Brzozowski's algorithm stands quite isolated and fits in neither of these two classes.

The algorithm for the minimization of complete deterministic finite state automata given by Hopcroft [34] runs in worst-case time $O(n \log n)$. It is, up to now, the most efficient algorithm known in the general case. It has recently been extended to incomplete deterministic finite automata [50],[9].

Hopcroft's algorithm is related to Moore's partition refinement algorithm [44], although it is different. One of the purposes of this text is the comparison of the nature of Moore's and Hopcroft's algorithms. This gives some new insight into both algorithms. As we shall see, these algorithms are quite different both in behavior and in complexity. In particular, we show that it is not possible to simulate the computations of one algorithm by the other.

Moore's partition refinement algorithm is much simpler than Hopcroft's algorithm. It has been shown [7] that, although its worst-case behavior is quadratic, its average running time is $O(n \log n)$. No evaluation of the average is known for Hopcroft's algorithm.

The family of algorithms based on fusion of states is important in practice for the construction of minimal automata representing finite sets, such as dictionaries in natural language processing. A linear time implementation of such an algorithm for cycle-free automata was given by Revuz [47]. This algorithm has been extended to a more general class of automata by Almeida and Zeitoun [3], namely to automata where all strongly connected components are simple cycles. It has been demonstrated in [8] that minimization by state fusion, which is not always possible, works well for local automata.

There is another efficient incremental algorithm for finite sets, by Daciuk *et al.* [27]. The advantage of this algorithm is that it does not build the intermediate trie which is rather space consuming.

We also consider updating a minimal automaton when a word is added or removed from the set it recognizes.

Finally, we discuss briefly the case of nondeterministic automata. It is well-known that minimal nondeterministic automata are not unique. However, there are several subclasses where the minimal automaton is unique.

We do not consider here the problem of constructing a minimal automaton starting from another description of the regular language, such as the synthesis of an automaton from a regular expression. We also do not consider devices that may be more space efficient, such as alternating automata or two-way automata. Other cases not considered here concern sets of infinite words and the minimization of their accepting devices.

The chapter is organized as follows. The first section just fixes notation, the next describes briefly Brzozowski's algorithm. In Section 4, we give basic facts on Moore's minimization algorithm. Section 5 is a detailed description of Hopcroft's algorithm, with the proof of correctness and running time. It also contains the comparison of Moore's and Hopcroft's algorithms. The next section is devoted to so-called slow automata. Some material in these two sections is new.

Sections 7 and 8 are devoted to the family of algorithms working by fusion. We describe in particular Revuz's algorithm and its generalization by Almeida and Zeitoun, the incremental algorithm of Daciuk *et al.*, and dynamic minimization. The last section contains miscellaneous results on special cases and a short discussion of nondeterministic minimal automata.

2 Definitions and notation

It appears to be useful, for a synthetic presentation of the minimization algorithms of Moore and of Hopcroft, to introduce some notation for partitions of the set of states. This section just fixes this notation.

Partitions and equivalence relations. A *partition* of a set E is a family $\mathcal{P}$ of nonempty, pairwise disjoint subsets of E such that $E = \bigcup_{P \in \mathcal{P}} P$. The *index* of the partition is the number of its elements. A partition defines an equivalence relation $\equiv_{\mathcal{P}}$ on E. Conversely, the set of all equivalence classes $[x]$, for $x \in E$, of an equivalence relation on E defines a partition of E. This is the reason why all terms defined for partitions have the same meaning for equivalence relations and vice versa.

A subset F of E is *saturated* by $\mathcal{P}$ if it is the union of classes of $\mathcal{P}$. Let $\mathcal{Q}$ be another partition of E. Then $\mathcal{Q}$ is a *refinement* of $\mathcal{P}$, or $\mathcal{P}$ is *coarser* than $\mathcal{Q}$, if each class of $\mathcal{Q}$ is contained in some class of $\mathcal{P}$. If this holds, we write $\mathcal{Q} \leqslant \mathcal{P}$. The index of $\mathcal{Q}$ is then larger than the index of $\mathcal{P}$.

Given two partitions $\mathcal{P}$ and $\mathcal{Q}$ of a set E, we denote by $\mathcal{U} = \mathcal{P} \wedge \mathcal{Q}$ the coarsest partition which refines $\mathcal{P}$ and $\mathcal{Q}$. The classes of $\mathcal{U}$ are the nonempty sets $P \cap Q$, for

$P \in \mathcal{P}$ and $Q \in \mathcal{Q}$. The notation is extended to a set of partitions in the usual way: we write $\mathcal{P} = \mathcal{P}_1 \wedge \cdots \wedge \mathcal{P}_n$ for the common refinement of $\mathcal{P}_1, \ldots, \mathcal{P}_n$. If $n = 0$, then $\mathcal{P}$ is the universal partition of E composed of the single class E. This partition is the neutral element for the $\wedge$-operation.

Let F be a subset of E. A partition $\mathcal{P}$ of E induces a partition $\mathcal{P}'$ of F by intersection: $\mathcal{P}'$ is composed of the nonempty sets $P \cap F$, for $P \in \mathcal{P}$. If $\mathcal{P}$ and $\mathcal{Q}$ are partitions of E and $\mathcal{Q} \leqslant \mathcal{P}$, then the restrictions $\mathcal{P}'$ and $\mathcal{Q}'$ to F still satisfy $\mathcal{Q}' \leqslant \mathcal{P}'$.

If $\mathcal{P}$ and $\mathcal{P}'$ are partitions of disjoint sets E and E', we denote by $\mathcal{P} \vee \mathcal{P}'$ the partition of $E \cup E'$ whose restriction to E and E' are $\mathcal{P}$ and $\mathcal{P}'$ respectively. So, one may write

$$\mathcal{P} = \bigvee_{P \in \mathcal{P}} \{P\}\,.$$

Minimal automaton. We consider a deterministic automaton $\mathcal{A} = (Q, i, F)$ over the alphabet A with set of states Q, initial state i, and set of final states F. To each state q corresponds a subautomaton of $\mathcal{A}$ obtained when q is chosen as the initial state. We call it the *subautomaton rooted at* q or simply the automaton at q. Usually, we consider only the trim part of this automaton. To each state q corresponds a language $L_q(\mathcal{A})$ which is the set of words recognized by the subautomaton rooted at q, that is

$$L_q(\mathcal{A}) = \{w \in A^* \mid q \cdot w \in F\}\,.$$

This language is called the *future* of the state q, or also the *right language* of this state. Similarly one defines the *past* of q, also called the *left language*, as the set $\{w \in A^* \mid i \cdot w = q\}$. The automaton $\mathcal{A}$ is *minimal* if $L_p(\mathcal{A}) \neq L_q(\mathcal{A})$ for each pair of distinct states p, q. The equivalence relation $\equiv$ defined by

$$p \equiv q \quad \text{if and only if} \quad L_p(\mathcal{A}) = L_q(\mathcal{A})$$

is a *congruence*, that is $p \equiv q$ implies $p \cdot a \equiv q \cdot a$ for all letters a. It is called the *Nerode congruence*. Note that the Nerode congruence saturates the set of final states. Thus an automaton is minimal if and only if its Nerode equivalence is the identity.

Minimizing an automaton is the problem of computing the Nerode equivalence. Indeed, the *quotient* automaton $\mathcal{A}/{\equiv}$ obtained by taking for set of states the set of equivalence classes of the Nerode equivalence, for the initial state the class of the initial state i, for set of final states the set of equivalence classes of states in F and by defining the transition function by $[p] \cdot a = [p \cdot a]$ accepts the same language, and its Nerode equivalence is the identity. The minimal automaton recognizing a given language is unique.

Partitions and automata. Again, we fix a deterministic automaton $\mathcal{A} = (Q, i, F)$ over the alphabet A. It is convenient to use the shorthand P^c for $Q \setminus P$ when P is a subset of the set Q.

Given a set $P \subset Q$ of states and a letter a, we denote by $a^{-1}P$ the set of states q such that $q \cdot a \in P$. Given sets $P, R \subset Q$ and $a \in A$, we denote by

$$(P, a)|R$$

the partition of R composed of the nonempty sets among the two sets

$$R \cap a^{-1}P = \{q \in R \mid q \cdot a \in P\} \quad \text{and} \quad R \setminus a^{-1}P = \{q \in R \mid q \cdot a \notin P\}\,.$$

Note that $R \setminus a^{-1}P = R \cap (a^{-1}P)^c = R \cap a^{-1}(P^c)$ so the definition is symmetric in P and P^c. In particular

$$(P, a)|R = (P^c, a)|R\,. \tag{2.1}$$

The pair (P, a) is called a *splitter*. Observe that $(P, a)|R = \{R\}$ if either $R \cdot a \subset P$ or $R \cdot a \cap P = \emptyset$, and $(P, a)|R$ is composed of two classes if both $R \cdot a \cap P \neq \emptyset$ and $R \cdot a \cap P^c \neq \emptyset$ or equivalently if $R \cdot a \not\subset P^c$ and $R \cdot a \not\subset P$. If $(P, a)|R$ contains two classes, then we say that (P, a) *splits* R. Note that the pair $S = (P, a)$ is called a splitter even if it does not split.

It is useful to extend the notation above to words. Given a word w and sets $P, R \subset Q$ of states, we denote by $w^{-1}P$ the set of states such that $q \cdot w \in P$, and by $(P, w)|R$ the partition of R composed of the nonempty sets among

$$R \cap w^{-1}P = \{q \in R \mid q \cdot w \in P\} \quad \text{and} \quad R \setminus w^{-1}P = \{q \in R \mid q \cdot w \notin P\}\,.$$

As an example, the partition $(F, w)|Q$ is the partition of Q into the set of those states from which w is accepted, and the other ones. A state q in one of the sets and a state q' in the other are sometimes called *separated* by w.

The Nerode equivalence is the coarsest equivalence relation on the set of states that is a (right) congruence saturating F. With the notation of splitters, this can be rephrased as follows.

Proposition 2.1. *The partition corresponding to the Nerode equivalence is the coarsest partition $\mathcal{P}$ such that no splitter (P, a), with $P \in \mathcal{P}$ and $a \in A$, splits a class in $\mathcal{P}$, that is such that $(P, a)|R = \{R\}$ for all $P, R \in \mathcal{P}$ and $a \in A$.* □

We use later the following lemma which is already given in Hopcroft's paper [34]. It is the basic observation that ensures that Hopcroft's algorithm works correctly.

Lemma 2.2. *Let P be a set of states, and let $\mathcal{P} = \{P_1, P_2\}$ be a partition of P. For any letter a and for any set of states R, one has*

$$(P, a)|R \wedge (P_1, a)|R = (P, a)|R \wedge (P_2, a)|R = (P_1, a)|R \wedge (P_2, a)|R\,,$$

and consequently

$$(P, a)|R \geqslant (P_1, a)|R \wedge (P_2, a)|R\,, \tag{2.2}$$

$$(P_1, a)|R \geqslant (P, a)|R \wedge (P_2, a)|R\,. \tag{2.3}$$

3 Brzozowski's algorithm

The minimization algorithm given by Brzozowski [18] is quite different from the two families of iterative algorithms (by refinement and by fusion) that we consider in this chapter. Although its worst-case behavior is exponential, it is conceptually simple, easy

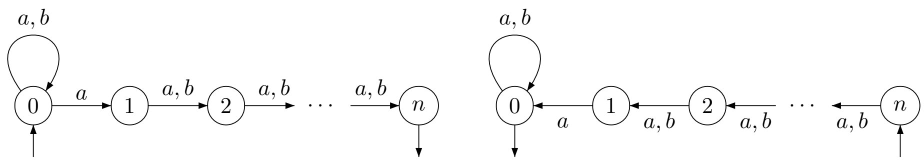

Figure 1. The automaton on the left recognizing the language A^*aA^n. It has $n+1$ states and the minimal deterministic automaton for this language has 2^n states. The automaton on the right is its reversal. It is minimal and recognizes A^naA^*.

to implement, and it is quite efficient in many cases. Moreover, it does not require the automaton to be deterministic, contrary to the algorithms described later.

Given an automaton $\mathcal{A} = (Q, I, F, E)$ over an alphabet A, its *reversal* is the automaton denoted A^R obtained by exchanging the initial and the final states, and by inverting the orientation of the edges. Formally $\mathcal{A}^R = (Q, F, I, E^R)$, where $E^R = \{(p, a, q) \mid (q, a, p) \in E\}$. The basis for Brzozowski's algorithm is the following result.

Proposition 3.1. *Let $\mathcal{A}$ be a finite deterministic automaton, and let $\mathcal{A}^\sim$ be the deterministic trim automaton obtained by determinizing and trimming the reversal $\mathcal{A}^R$. Then $\mathcal{A}^\sim$ is minimal.*

For a proof of this proposition, see for instance Sakarovitch's book [48]. The minimization algorithm now is just a double application of the operation. Observe that the automaton one starts with need not to be deterministic.

Corollary 3.2. *Let $\mathcal{A}$ be a finite automaton. Then $(\mathcal{A}^\sim)^\sim$ is the minimal automaton of $\mathcal{A}$.*

Example 3.1. We consider the automata given in Figure 1 over the alphabet $A = \{a, b\}$. Each automaton is the reversal of the other. However, determinization of the automaton on the left requires exponential time and space.

4 Moore's algorithm

The minimization algorithm given by Moore [44] computes the Nerode equivalence by a stepwise refinement of some initial equivalence. All automata are assumed to be deterministic.

4.1 Description

Let $\mathcal{A} = (Q, i, F)$ be an automaton over an alphabet A. Define, for $q \in Q$ and $h \geqslant 0$, the set

$$L_q^{(h)}(\mathcal{A}) = \{w \in A^* \mid |w| \leqslant h,\ q \cdot w \in F\}\,.$$

The *Moore equivalence* of order h is the equivalence $\equiv_h$ on Q defined by

$$p \equiv_h q \iff L_p^{(h)}(\mathcal{A}) = L_q^{(h)}(\mathcal{A})\,.$$

MOORE($\mathcal{A}$)
  $\mathcal{P} \leftarrow \{F, F^c\}$ $\triangleright$ The initial partition
  repeat
    $\mathcal{P}' \leftarrow \mathcal{P}$ $\triangleright$ $\mathcal{P}'$ is the old partition, $\mathcal{P}$ is the new one
    for all $a \in A$ **do**
      $\mathcal{P}_a \leftarrow \bigwedge_{P\in\mathcal{P}}(P,a)|Q$
    $\mathcal{P} \leftarrow \mathcal{P} \wedge \bigwedge_{a\in A}\mathcal{P}_a$
  until $\mathcal{P} = \mathcal{P}'$

Figure 2. Moore's minimization algorithm.

Using the notation of partitions introduced above, one can rephrase the definitions of the Nerode equivalence and of the Moore equivalence of order h. These are the equivalences defined by

$$\bigwedge_{w\in A^*}(F,w)|Q\,,\quad \text{and} \quad \bigwedge_{w\in A^*,\ |w|\leqslant h}(F,w)|Q\,.$$

Since the set of states is finite, there is a smallest h such that the Moore equivalence $\equiv_h$ equals the Nerode equivalence $\equiv$. We call this integer the *depth* of Moore's algorithm on the finite automaton $\mathcal{A}$, or the depth of $\mathcal{A}$ for short. The depth depends in fact only on the language recognized by the automaton, and not on the particular automaton under consideration. Indeed, each state of an automaton recognizing a language L represents in fact a left quotient $u^{-1}L$ for some word u.

The depth is the smallest h such that $\equiv_h$ equals $\equiv_{h+1}$. This leads to the refinement algorithm that computes successively $\equiv_0$, $\equiv_1$, $\ldots$, $\equiv_h$, $\ldots$, halting as soon as two consecutive equivalences are equal. The next property gives a method to compute the Moore equivalences efficiently.

Proposition 4.1. *For two states $p, q \in Q$, and $h \geqslant 0$, one has*

$$p \equiv_{h+1} q \quad \Longleftrightarrow \quad p \equiv_h q \ \textit{and}\ \ p\cdot a \equiv_h q\cdot a \ \textit{for all}\ a \in A\,. \tag{4.1}$$

We use this proposition in a slightly different formulation. Denote by $\mathcal{M}_h$ the partition corresponding to the Moore equivalence $\equiv_h$. Then the following equations hold.

Proposition 4.2. *For $h \geqslant 0$, one has*

$$\mathcal{M}_{h+1} = \mathcal{M}_h \wedge \bigwedge_{a\in A}\ \bigwedge_{P\in\mathcal{M}_h}(P,a)|Q \ = \bigvee_{R\in\mathcal{M}_h}\Big(\bigwedge_{a\in A}\ \bigwedge_{P\in\mathcal{M}_h}(P,a)|R\Big)\,. \qquad \square$$

The computation is described in Figure 2. It is realized by a loop that refines the current partition. The computation of the refinement of k partitions of a set swith n elements can be done in time $O(kn^2)$ by brute force. A radix sort improves the running time to $O(kn)$. With $k = \mathrm{Card}(A)$, each tour in the loop is realized in time $O(kn)$, so the total time is $O(\ell kn)$, where ℓ is the number of refinement steps in the computation of the Nerode equivalence $\equiv$, that is the depth of the automaton.

The worst case behavior is obtained for $\ell = n - 2$. We say that automata having maximal depth are *slow* and more precisely are *slow for Moore* automata. These automata are investigated later. We will show that they are equivalent to automata we call *slow for Hopcroft*.

4.2 Average complexity

The average case behavior of Moore's algorithm has recently been studied in several papers. We report here some results given in [7, 28]. The authors make a detailed analysis of the distribution of the number ℓ of refinement steps in Moore's algorithm, that is of the depth of automata, and they prove that there are only a few automata for which this depth is larger than $\log n$.

More precisely, fix some alphabet and we consider deterministic automata over this alphabet.

A *semi-automaton* $\mathcal{K}$ is an automaton whose set of final states is not specified. Thus, an automaton is a pair $(\mathcal{K}, F)$, where $\mathcal{K}$ is a semi-automaton and F is the set of final states. The following theorem given an upper bound on the average complexity of Moore's algorithm for all automata derived from a given semiautomaton.

Theorem 4.3 (Bassino, David, Nicaud [7]). *Let $\mathcal{K}$ be a semi-automaton with n states. The average complexity of Moore's algorithm for the automata $(\mathcal{K}, F)$, for the uniform probability distribution over the sets F of final states, is $O(n \log n)$.*

The result also holds for Bernoulli distributions for final states. The result remains valid for subfamilies of automata such as strongly connected automata or group automata.

When all semi-automata are considered to be equally like, then the following bound is valid.

Theorem 4.4 (David [28]). *The average complexity of Moore's algorithm, for the uniform probability over all complete automata with n states, is $O(n \log \log n)$.*

This result is remarkable in view of the lower bound which is given in the following statement [6].

Theorem 4.5. *If the underlying alphabet has at least two letters, then Moore's algorithm, applied on a minimal automaton with n states, requires at least $\Omega(n \log \log n)$ operations.*

5 Hopcroft's algorithm

Hopcroft [34] has given an algorithm that computes the minimal automaton of a given deterministic automaton. The running time of the algorithm is $O(k\, n \log n)$ where k is the cardinality of the alphabet and n is the number of states of the given automaton. The algorithm has been described and re-described several times [32, 1, 10, 16, 40].

HOPCROFT($\mathcal{A}$)

$\mathcal{P} \leftarrow \{F, F^c\}$ ▷ The initial partition
$\mathcal{W} \leftarrow \emptyset$ ▷ The waiting set
for all $a \in A$ **do**
    ADD($(\min(F, F^c), a), \mathcal{W}$) ▷ Initialization of the waiting set
while $\mathcal{W} \neq \emptyset$ **do**
    $(W, a) \leftarrow$ TAKESOME($\mathcal{W}$) ▷ Take and remove some splitter
    for each $P \in \mathcal{P}$ which is split by (W, a) **do**
        $P', P'' \leftarrow (W, a)|P$ ▷ Compute the split
        REPLACE P by P' and P'' in $\mathcal{P}$ ▷ Refine the partition
        for all $b \in A$ **do** ▷ Update the waiting set
            if $(P, b) \in \mathcal{W}$ **then**
                REPLACE (P, b) by (P', b) and (P'', b) in $\mathcal{W}$
            else
                ADD($(\min(P', P''), b), \mathcal{W}$)

Figure 3. Hopcroft's minimization algorithm.

5.1 Outline

The algorithm is outlined in the function HOPCROFT given in Figure 3. We denote by $\min(P, P')$ the set of smaller size of the two sets P and P', and any one of them if they have the same size.

Given a deterministic automaton $\mathcal{A}$, Hopcroft's algorithm computes the coarsest congruence which saturates the set F of final states. It starts from the partition $\{F, F^c\}$ which obviously saturates F and refines it until it gets a congruence. These refinements of the partition are always obtained by splitting some class into two classes.

The algorithm proceeds as follows. It maintains a current partition $\mathcal{P} = \{P_1, \ldots, P_n\}$ and a current set $\mathcal{W}$ of *splitters*, that is of pairs (W, a) that remain to be processed, where W is a class of $\mathcal{P}$ and a is a letter. The set $\mathcal{W}$ is called the *waiting* set. The algorithm stops when the waiting set $\mathcal{W}$ becomes empty. When it stops, the partition $\mathcal{P}$ is the coarsest congruence that saturates F. The starting partition is the partition $\{F, F^c\}$ and the starting set $\mathcal{W}$ contains all pairs $(\min(F, F^c), a)$ for $a \in A$.

The main loop of the algorithm removes one splitter (W, a) from the waiting set $\mathcal{W}$ and performs the following actions. Each class P of the current partition (including the class W) is checked as to whether it is split by the pair (W, a). If (W, a) does not split P, then nothing is done. On the other hand, if (W, a) splits P into say P' and P'', the class P is replaced in the partition $\mathcal{P}$ by P' and P''. Next, for each letter b, if the pair (P, b) is in $\mathcal{W}$, it is replaced in $\mathcal{W}$ by the two pairs (P', b) and (P'', b), otherwise only the pair $(\min(P', P''), b)$ is added to $\mathcal{W}$.

It should be noted that the algorithm is not really deterministic because it has not been specified which pair (W, a) is taken from $\mathcal{W}$ to be processed at each iteration of the main loop. This means that for a given automaton, there are many executions of the algorithm. It turns out that all of them produce the right partition of the states. However, different executions may give rise to different sequences of splitting and also to different running time. Hopcroft has proved that the running time of any execution is bounded by $O(|A|n\log n)$.

5.2 Behavior

The pair $(\mathcal{P},\mathcal{W})$ composed of the current partition and the current waiting set in some execution of Hopcroft's algorithm is called a *configuration*. The following proposition describes the evolution of the current partition in Hopcroft's algorithm. Formula 5.1 is the key inequality for the proofs of correctness and termination. We will use it in the special case where the set R is a class of the current partition.

Proposition 5.1. *Let $(\mathcal{P},\mathcal{W})$ be a configuration in some execution of Hopcroft's algorithm on an automaton $\mathcal{A}$ on A. For any $P\in\mathcal{P}$, any subset R of a class of $\mathcal{P}$, and $a\in A$, one has*

$$(P,a)|R \geqslant \bigwedge_{(W,a)\in\mathcal{W}} (W,a)|R\,, \tag{5.1}$$

that is, the partition $(P,a)|R$ is coarser than the partition $\bigwedge_{(W,a)\in\mathcal{W}}(W,a)|R$.

Proof. The proof is by induction on the steps of an execution. The initial configuration $(\mathcal{P},\mathcal{W})$ is composed of the initial partition is $\mathcal{P}=\{F,F^c\}$ and the initial waiting set $\mathcal{W}$ is either $\mathcal{W}=\{(F,a)\mid a\in A\}$ or $\mathcal{W}=\{(F^c,a)\mid a\in A\}$. Since the partitions $(F,a)|R$ and $(F^c,a)|R$ are equal for any set R the proposition is true.

Assume now that $(\mathcal{P},\mathcal{W})$ is not the initial configuration. Let $(\widehat{\mathcal{P}},\widehat{\mathcal{W}})$ be a configuration that precedes immediately $(\mathcal{P},\mathcal{W})$. Thus $(\mathcal{P},\mathcal{W})$ is obtained from $(\widehat{\mathcal{P}},\widehat{\mathcal{W}})$ in one step of Hopcroft's algorithm, by choosing one splitter S in $\widehat{\mathcal{W}}$, and by performing the required operations on $\widehat{\mathcal{P}}$ and $\widehat{\mathcal{W}}$.

First we observe that, by the algorithm, and by Lemma 2.2, one has for any set of states R,

$$\bigwedge_{(\widehat{W},a)\in\widehat{\mathcal{W}}\setminus\{S\}} (\widehat{W},a)|R \geqslant \bigwedge_{(W,a)\in\mathcal{W}} (W,a)|R\,. \tag{5.2}$$

Indeed, the set $\mathcal{W}$ contains all $\widehat{\mathcal{W}}\setminus\{S\}$ with the exception of those pairs (P,a) for which P is split into two parts, and in this case, the relation follows from (2.2).

Next, consider a subset R of a set of $\mathcal{P}$. Since R was not split by S, that is since $S|R=\{R\}$, we have

$$\bigwedge_{(\widehat{W},a)\in\widehat{\mathcal{W}}} (\widehat{W},a)|R = \bigwedge_{(\widehat{W},a)\in\widehat{\mathcal{W}}\setminus\{S\}} (\widehat{W},a)|R\,. \tag{5.3}$$

Moreover, by the induction hypothesis, and since R is also a subset of a set of $\widehat{\mathcal{P}}$, we have for any $\widehat{P}\in\widehat{\mathcal{P}}$,

$$(\widehat{P},a)|R \geqslant \bigwedge_{(\widehat{W},a)\in\widehat{\mathcal{W}}} (\widehat{W},a)|R\,. \tag{5.4}$$

Consequently, for any subset R of a set of $\mathcal{P}$, any $\widehat{P} \in \widehat{\mathcal{P}}$, in view of 5.2, 5.3 and 5.4, we have

$$(\widehat{P}, a)|R \geqslant \bigwedge_{(W,a)\in\mathcal{W}} (W, a)|R\,. \tag{5.5}$$

Let now $R \in \mathcal{P}$, $a \in A$, and let again R be a subset of a set of $\mathcal{P}$. We consider the partition $(P, a)|R$. We distinguish two cases.

Case 1. Assume $P \in \widehat{\mathcal{P}}$. The proof follows directly from (5.5).

Case 2. Assume $P \notin \widehat{\mathcal{P}}$. Then there exists $\widehat{P} \in \widehat{\mathcal{P}}$ such that $S|\widehat{P} = \{P, P'\}$.

If $(\widehat{P}, a) \in \widehat{\mathcal{W}} \setminus \{S\}$, then both (P, a) and (P', a) are in $\mathcal{W}$. Consequently

$$(P, a)|R \geqslant \bigwedge_{(W,a)\in\mathcal{W}} (W, a)|R\,.$$

If, on the contrary, $(\widehat{P}, a) \notin \widehat{\mathcal{W}} \setminus \{S\}$, then by the algorithm, one of (P, a) or (P', a) is in $\mathcal{W}$.

Case 2a. Assume that $(P, a) \in \mathcal{W}$. Then obviously

$$(P, a)|R \geqslant \bigwedge_{(W,a)\in\mathcal{W}} (W, a)|R\,.$$

Case 2b. Assume $(P', a) \in \mathcal{W}$. By Lemma 2.2, we have

$$(P, a)|R \geqslant (P', a)|R \wedge (\widehat{P}, a)|R$$

as obviously we have

$$(P', a)|R \geqslant \bigwedge_{(W,a)\in\mathcal{W}} (W, a)|R\,.$$

and by use of (5.5), we obtain

$$(P, a)|R \geqslant \bigwedge_{(W,a)\in\mathcal{W}} (W, a)|R\,.$$

This completes the proof. ☐

Corollary 5.2. *The current partition at the end of an execution of Hopcroft's algorithm on an automaton $\mathcal{A}$ is the Nerode partition of $\mathcal{A}$.*

Proof. Let $\mathcal{P}$ be the partition obtained at the end of an execution of Hopcroft's algorithm on an automaton $\mathcal{A}$. By Proposition 2.1, it suffices to check that no splitter splits a class of $\mathcal{P}$. Since the waiting set $\mathcal{W}$ is empty, the right-hand side of (5.1) evaluates to $\{R\}$ for each triple (P, a, R). This means that (P, a) indeed does not split R. ☐

5.3 Complexity

Proposition 5.3. *Hopcroft's algorithm can be implemented to have worst-case time complexity $O(kn \log n)$ for an automaton with n states over a k-letter alphabet.*

To achieve the bound claimed, a partition $\mathcal{P}$ of a set Q should be implemented in a way to allow the following operations:

- accessing the class to which a state belongs in constant time;
- enumeration of the elements of a class in time proportional to its size;
- adding and removing of an element in a class in constant time.

The computation of all splittings P', P'' of classes P by a given splitter (W, a) is done in time $O(\mathrm{Card}(a^{-1}W))$ as follows.

(1) One enumerates the states q in $a^{-1}W$. For each state q, the class P of q is marked as a candidate for splitting, the state q is added to a list of states to be removed from P, and a counter for the number of states in the list is incremented.
(2) Each class that is marked is a candidate for splitting. It is split if the number of states to be removed differs from the size of the class. If this holds, the states in the list of P are removed to build a new class. The other states remain in P.

The waiting set $\mathcal{W}$ is implemented such that membership can be tested in constant time, and splitters can be added and removed in constant time. This allows the replacement of a splitter (P, b) by the two splitters (P', b) and (P'', b) in constant time, since in fact P' is just the modified class P, and it suffices to add the splitter (P'', b).

Several implementations of partitions that satisfy the time requirements exist. Hopcroft [34] describes such a data structure, reported in [14]. Knuutila [40] gives a different implementation.

Proof of Proposition 5.3. For a given state q, a splitter (W, a) such that $q \in W$ is called a *q-splitter*.

Consider some q-splitter. When it is removed from the waiting set $\mathcal{W}$, it may be smaller than when it was added, because it may have been split during its stay in the waiting set. On the contrary, when a q-splitter is added to the waiting set, then its size is at most one half of the size it had when it was previously removed. Thus, for a fixed state q, the number of q-splitters (W, a) which are removed from $\mathcal{W}$ is at most $k \log n$, since at each removal, the number of states in W is at most one half of the previous addition.

The total number of elements of the sets $a^{-1}W$, where (W, a) is in $\mathcal{W}$, is $O(kn \log n)$. Indeed, for a fixed state $q \in Q$, a state p such that $p \cdot a = q$ is exactly in those sets $a^{-1}W$, where (W, a) is a q-splitter in $\mathcal{W}$. There are at most $O(\log n)$ of such sets for each letter a, so at most $(k \log n)$ sets for each fixed q. Since there are n states, the claim follows. This completes the proof since the running time is bounded by the size of the sets $a^{-1}W$, where (W, a) is in $\mathcal{W}$. $\square$

5.4 Miscellaneous remarks

There are some degrees of freedom in Hopcroft's algorithm. In particular, the way the waiting set is represented may influence the efficiency of the algorithm. This issue has been considered in [40]. In [5] some practical experiments are reported. In [45] it is shown that the worst-case reported in [13] in the case of de Bruijn words remains of this complexity when the waiting set is implemented as a queue (LIFO), whereas this complexity is never reached with an implementation as a stack (FIFO). See this paper for other discussions, in particular in relation with cover automata.

	0	1	2	3	4	5	6	7	8	9
a	2	1	3	3	5	7	8	1	7	6
b	9	1	5	7	8	7	8	1	7	4

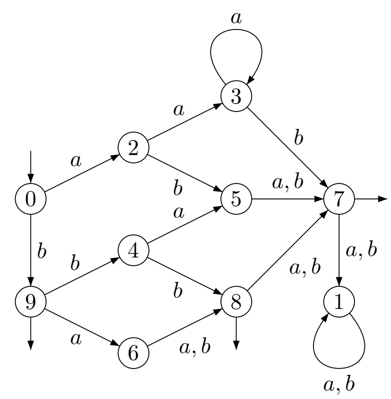

Figure 4. Next-state function of the example automaton.

Hopcroft's algorithm, as reported here, requires the automaton to be complete. This may be a serious drawback in the case where the automaton has only a few transitions. For instance, if a dictionary is represented by a trie, then the average number of edges per state is about 1.5 for the French dictionary (personal communication of Dominique Revuz), see Table 2 below. Recently, two generalizations of Hopcroft's algorithm to incomplete automata were presented, by [9] and [50], with running time $O(m \log n)$, where n is the number of states and m is the number of transitions. Since $m \leqslant kn$ where k is the size of the alphabet, the algorithm achieves the same time complexity.

5.5 Moore versus Hopcroft

We present an example which illustrates the fact that Hopcroft's algorithm is not just a refinement of Moore's algorithm. This is proved by checking that one of the partitions computed by Moore's algorithm in the example does not appear as a partition in any of the executions of Hopcroft's algorithm on this automaton.

The automaton we consider is over the alphabet $A = \{a, b\}$. Its set of states is $Q = \{0, 1, 2, \ldots, 9\}$, the set of final states is $F = \{7, 8, 9\}$. The next-state function and the graph are given in Figure 4.

The Moore partitions are easily computed. The partition $\mathcal{M}_1$ of order 1 is composed of the five classes:

$$\{0, 3, 4\}, \{1, 2\}, \{5, 6\}, \{7, 9\}, \{8\}\,.$$

The Moore partition of order 2 is the identity.

The initial partition for Hopcroft's algorithm is $\{F, F^c\}$, where $F = 789$ (we will represent a set of states by the sequence of its elements). The initial waiting set is composed of $(789, a)$ and $(789, b)$. There are two cases, according to the choice of the first splitter.

Case 1. The first splitter is $(789, a)$. Since $a^{-1}789 = 568$, each of the classes F and F^c is split. The new partition is $01234|56|79|8$. The new waiting set is

$$(79, b), (8, b), (8, a), (56, a), (56, b)\,.$$

P	56	8	79	12
$a^{-1}P$	49	6	58	017
$b^{-1}P$	2	46	0357	17

Table 1. The sets $c^{-1}P$, with (P, c) in a waiting set.

The first three columns in Table 1 contain the sets $c^{-1}P$, for (P, c) in this waiting set. By inspection, one sees that each entry in these columns cuts off at least one singleton class which is not in the Moore equivalence $\mathcal{M}_1$. This implies that $\mathcal{M}_1$ cannot be obtained by Hopcroft's algorithm in this case.

Case 2. The first splitter is $(789, b)$. Since $b^{-1}789 = 034568$, the new partition is $12|03456|79|8$. The new waiting set is

$$(79, a), (8, a), (8, b), (12, a), (12, b)\,.$$

Again, each entry in the last three columns of Table 1 cuts off at least one singleton class which is not in the Moore equivalence $\mathcal{M}_1$. This implies that, also in this case, $\mathcal{M}_1$ cannot be obtained by Hopcroft's algorithm.

Despite the difference illustrated by this example, there are similarities between Moore's and Hopcroft's algorithms that have been exploited by Julien David in his thesis [29] to give an upper bound on the average running time of Hopcroft's algorithm for a particular strategy.

In this strategy, there are two waiting sets, the current set $\mathcal{W}$ and a *future waiting set* $\mathcal{F}$. Initially, $\mathcal{F}$ is empty. Hopcroft's algorithm works as usual, except for line 14: Here, the splitter $(\min(P', P''), b)$ is added to $\mathcal{F}$ and not to $\mathcal{W}$. When $\mathcal{W}$ is empty, then the contents of $\mathcal{F}$ and $\mathcal{W}$ are swapped. The algorithm stops when both sets $\mathcal{W}$ and $\mathcal{F}$ are empty.

Proposition 5.4 (David [29]). *There is a strategy for Hopcroft's algorithm such that its average complexity, for the uniform probability over all complete automata with n states, is $O(n \log\log n)$.*

Julien David shows that at the end of each cycle, that is when $\mathcal{W}$ becomes empty, the current partition $\mathcal{P}$ of the set of states is in fact a refinement of the corresponding level in Moore's algorithm. This shows that the number of cycles in Hopcroft's algorithm, for this strategy, is bounded by the depth of the automaton. Thus Theorem 4.4 applies.

6 Slow automata

We are concerned in this section with automata that behave badly for Hopcroft's and Moore's minimization algorithms. In other terms, we look for automata for which Moore's algorithm requires the maximal number of steps, and similarly for Hopcroft's algorithm.

6.1 Definition and equivalence

Recall that an automaton with n states is called *slow for Moore* if the number ℓ of steps in Moore's algorithm is $n-2$. A slow automaton is minimal. It is equivalent to say that each Moore equivalence $\equiv_h$ has exactly $h+2$ equivalence classes for $h \leqslant n-2$. This is due to the fact that, at each step, just one class of $\equiv_h$ is split, and that this class is split into exactly two classes of the equivalence $\equiv_{h+1}$.

Proposition 4.2 takes the following special form for slow automata.

Proposition 6.1. *Let $\mathcal{A}$ be an automaton with n states which is slow for Moore. For all $n-2 > h \geqslant 0$, there is exactly one class R in $\mathcal{M}_h$ which is split, and moreover, if (P,a) and (P',a') split R, with $P, P' \in \mathcal{M}_h$, then $(P,a)|R = (P',a')|R$.* □

An automaton is *slow for Hopcroft* if, for *all executions* of Hopcroft's algorithm, the splitters in the current waiting set either do not split or split in the same way: there is a unique class that is split into two classes, and always into the same two classes.

More formally, at each step $(\mathcal{W}, \mathcal{P})$ of an execution, there is at most one class R in the current partition $\mathcal{P}$ that is split, and for all splitters (P,a) and (P',a') in $\mathcal{W}$ that split R, one has $(P,a)|R = (P',a')|R$.

The definition is close to the statement in Proposition 6.1 above, and indeed, one has the following property.

Theorem 6.2. *An automaton is slow for Moore if and only if it is slow for Hopcroft.*

Proof. Let $\mathcal{A}$ be a finite automaton. We first suppose that $\mathcal{A}$ is slow for Moore. We consider an execution of Hopcroft's algorithm, and we prove that each step of the execution that changes the partition produces a Moore partition.

This holds for the initial configuration $(\mathcal{W}, \mathcal{P})$, since $\mathcal{P} = \mathcal{M}_0$. Assume that one has $\mathcal{P} = \mathcal{M}_h$ for some configuration $(\mathcal{W}, \mathcal{P})$ and some $h \geqslant 0$. Let R be the class of $\mathcal{M}_h$ split by Moore's algorithm.

Let $S \in \mathcal{W}$ be the splitter chosen in Hopcroft's algorithm. Then either S splits no class, and the partition remains equal to $\mathcal{M}_h$ or by Proposition 6.1 it splits the class R. In the second case, this class is split by S into two new classes, say R' and R''. The partition $\mathcal{P}' = \mathcal{P} \setminus \{R\} \cup \{R', R''\}$ is equal to $\mathcal{M}_{h+1}$.

Conversely, suppose that $\mathcal{A}$ is slow for Hopcroft. We show that it is also slow for Moore by showing that the partition $\mathcal{M}_{h+1}$ has only one class more than $\mathcal{M}_h$. For this, we use Proposition 4.2 which states that each class R in $\mathcal{M}_h$ is refined in $\mathcal{M}_{h+1}$ into the partition $\mathcal{R}$ given by

$$\mathcal{R} = \bigwedge_{a \in A} \bigwedge_{P \in \mathcal{M}_h} (P,a)|R\,.$$

We show by induction on the number of steps that, in any execution of Hopcroft's algorithm, $\mathcal{P} = \mathcal{M}_h$ for some configuration $(\mathcal{P}, \mathcal{W})$. This holds for the initial configuration. Let (W,a) be some splitter in $\mathcal{W}$. It follows from Proposition 5.1 that

$$\mathcal{R} \geqslant \bigwedge_{a \in A} \bigwedge_{(W,a) \in \mathcal{W}} (W,a)|R\,.$$

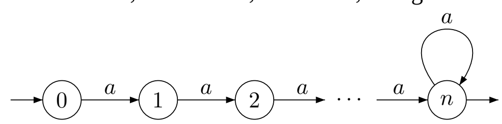

Figure 5. An automaton over one letter recognizing the set of words of length at least n.

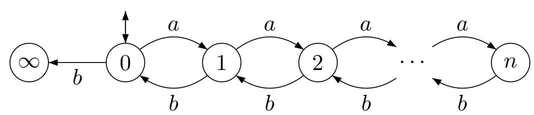

Figure 6. An automaton recognizing the Dyck words of "height" at most n.

Thus the partition $\mathcal{R}$ is coarser than that the partition of R obtained by Hopcroft's algorithm. Since the automaton is slow for Hopcroft, the partition on the right-hand side has at most two elements. More precisely, there is exactly one class R that is split by Hopcroft's algorithm into two classes. Since Moore's partition $\mathcal{M}_{h+1}$ is coarser, it contains precisely these classes. This proves the claim. $\square$

6.2 Examples

Example 6.1. The simplest example is perhaps the automaton given in Figure 5. For $0 \leqslant h \leqslant n-1$, the partition $\mathcal{M}_h$ is composed of the class $\{0, \ldots, n-h-1\}$, and of the singleton classes $\{n-h\}$, $\{n-h+1\}$,..., $\{n\}$. At each step, the last state is split off from the class $\{0, \ldots, n-h-1\}$.

Example 6.2. The automaton of Figure 6 recognizes the set $D^{(n)}$ of Dyck words w over $\{a, b\}$ such that $0 \leqslant |u|_a - |u|_b \leqslant n$ for all prefixes u of w. The partition $\mathcal{M}_h$, for $0 \leqslant h \leqslant n-1$, is composed of $\{0\}, \ldots, \{h\}$, and $\{\infty, h+1, \ldots, n\}$. At $h = n$, the state $\{\infty\}$ is separated from state n.

Example 6.3. Let $w = b_1 \cdots b_n$ be a word of length n over the binary alphabet $\{0, 1\}$. We define an automaton $\mathcal{A}_w$ over the unary alphabet $\{a\}$ as follows. The state set of $\mathcal{A}_w$ is $\{1, \ldots, n\}$ and the next state function is defined by $i \cdot a = i+1$ for $i < n$ and $n \cdot a = 1$. Note that the underlying labeled graph of $\mathcal{A}_w$ is just a cycle of length n. The final states really depend on w. The set of final states of $\mathcal{A}_w$ is $F = \{1 \leqslant i \leqslant n \mid b_i = 1\}$. We call such an automaton a *cyclic automaton*.

For a binary word u, we define Q_u to be the set of states of $\mathcal{A}_w$ which are the starting positions of circular occurrences of u in w. If u is the empty word, then Q_u is by convention the set Q of all states of $\mathcal{A}_w$. By definition, the set F of final states of $\mathcal{A}_w$ is Q_1 while its complement F^c is Q_0.

Consider the automaton $\mathcal{A}_w$ for $w = 01001010$ given in Fig. 7. The sets Q_1, Q_{01} and Q_{11} of states are respectively $\{2, 5, 7\}$, $\{1, 4, 6\}$ and $\emptyset$. If w is a Sturmian word, then the automaton $\mathcal{A}_w$ is slow.

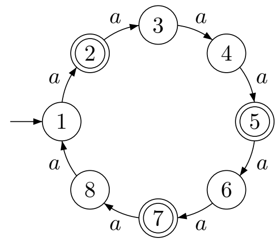

Figure 7. Cyclic automaton $\mathcal{A}_w$ for $w = 01001010$. Final states are circled.

Slow automata are closely related to binary Sturmian trees. Consider indeed a finite automaton, for instance over a binary alphabet. To this automaton corresponds an infinite binary tree, composed of all paths in the automaton. The nodes of the tree are labeled with the states encountered on the path. For each integer $h \geqslant 0$, the number of distinct subtrees of height h is equal to the number of classes in the Moore partition $\mathcal{M}_h$. It follows that the automaton is slow if and only if there are $h + 1$ distinct subtrees of height h for all h: this is precisely the definition of *Sturmian trees*, as given in [12].

We consider now the problem of showing that the running time $O(n \log n)$ for Hopcroft's algorithm on n-state automata is tight. The algorithm has a degree of freedom because, in each step of its main loop, it allows one to choose the splitter to be processed. Berstel and Carton [13] introduced a family of finite automata based on de Bruijn words. These are exactly the cyclic automata $\mathcal{A}_w$ of Example 6.3 where w is a binary de Bruijn word. They showed that there exist some "unlucky" sequence of choices that slows down the computation to achieve the lower bound $\Omega(n \log n)$.

In the papers [20] and [21], Castiglione, Restivo and Sciortino replace de Bruijn words by Fibonacci words. They observe that for these words, and more generally for all circular standard Sturmian words, there is no more choice in Hopcroft's algorithm. Indeed, the waiting set always contains only one element. The uniqueness of the execution of Hopcroft's algorithm implies by definition that the associated cyclic automata for Sturmian words are slow.

They show that, for Fibonacci words, the unique execution of Hopcroft's algorithm runs in time $\Omega(n \log n)$, so that the worst-case behavior is achieved for the cyclic automata of Fibonacci words. The computation is carried out explicitly, using connections between Fibonacci numbers and Lucas numbers. In [22], they give a detailed analysis of the reduction process that is the basis of their computation, and they show that this process is isomorphic, for all standard Sturmian words, to the refinement process in Hopcroft's algorithm.

In [11], the analysis of the running time of Hopcroft's algorithm is extended to cyclic automata of standard Sturmian words. It is shown that the directive sequences for which Hopcroft's algorithm has worst-case running time are those sequences $(d_1, d_2, d_3, \ldots)$ for which the sequence of geometric means $((p_n)^{1/n})_{n \geqslant 1}$, where $p_n = d_1 d_2 \cdots d_n$, is bounded.

7 Minimization by fusion

In this section, we consider the minimization of automata by fusion of states. An important application of this method is the computation of the minimal automaton recognizing a given finite set of words. This is widely used in computational linguistics for the space-efficient representation of dictionaries.

Let $\mathcal{A}$ be a deterministic automaton over the alphabet A, with set of states Q. The *signature* of a state p is the set of pairs $(a, q) \in A \times Q$ such that $p \cdot a = q$, together with a Boolean value denoting whether p is final or not. Two states p and q are called *mergeable* if and only if they have the same signature. The *fusion* or *merge* of two mergeable states p and q consists in replacing p and q by a single state. The state obtained by fusion of two mergeable states has the same signature.

Minimization of an automaton by a sequence of fusion of states with the same signature is not always possible. Consider the two-state automaton over the single letter a given in Figure 8 which recognizes a^*. It is not minimal. The signature of state 1 is $+, (a, 2)$ and the signature of state 2 is $+, (a, 1)$ (here "$+$" denotes an accepting state), so the states have different signatures and are not mergeable.

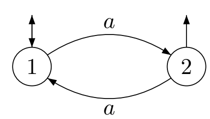

Figure 8. An automaton recognizing the set a^* which can not be minimized by fusion of its states.

7.1 Local automata

M.-P. Béal and M. Crochemore [8] designed an algorithm for minimizing a special class of deterministic automata by a sequence of state mergings. These automata are called irreducible *local* automata. They occur quite naturally in symbolic dynamics. The running time of the algorithm is $O(\min(m(n-r+1), m \log n))$, where m is the number of edges, n is the number of states, and r is the number of states of the minimized automaton. In particular, the algorithm is linear when the automaton is already minimal. Hopcroft's algorithm has running time $O(kn \log n)$, where k is the size of the alphabet, and since $kn \geqslant m$, it is worse than Béal and Crochemore's algorithm. Moreover, their algorithm does not require the automaton to be complete.

The automata considered here have several particular features. First, all states are both initial and final. Next, they are *irreducible*, that is, their underlying graph is strongly connected. Finally, the automata are *local*. By definition, this means that two distinct cycles carry different labels. This implies that the labels of a cycle are primitive words, since otherwise there exist different traversals of the cycle which have the same label. In [8], the constructions and proofs are done for a more general family of automata called AFT (for automata of *almost finite type*). We sketch here the easier case of local automata. Since all states are final, two states p and q of an automaton are *mergeable* if and only if, for all letters $a \in A$, $p \cdot a$ is defined if and only if $q \cdot a$ and, if this is the case, then $p \cdot a = q \cdot a$.

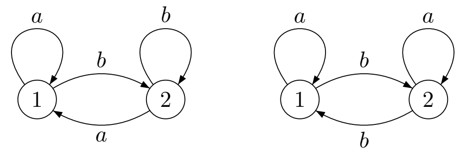

Figure 9. The automaton on the left is local, the automaton on the right is not because of the two loops labeled a, and because the label bb of the cycle through 1 and 2 is not a primitive word.

The basic proposition is the following. It shows that an irreducible local automaton can be minimized by a sequence of fusion of states.

Proposition 7.1. *If an irreducible local automaton is not minimal, then at least two of its states are mergeable.* □

The minimization algorithm assumes that the alphabet is totally ordered. It uses the notion of partial signature. First, with each state q is associated the signature $\sigma(q) = a_1p_1a_2p_2\cdots a_mp_m$, where $\{(a_1, p_1), \ldots, (a_m, p_m)\}$ is the signature of q, and the sequence is ordered by increasing value of the letters. Since all states are final, the Boolean indicator reporting this is omitted. A *partial signature* is any prefix $a_1p_1a_2p_2\cdots a_ip_i$ of a signature.

A first step consists in building a *signature tree* which represents the sets of states sharing a common partial signature. The root of the tree represents the set of all states, associated to the empty signature. A node representing the sets of states with a partial signature $a_1p_1a_2p_2\cdots a_ip_i$ is the parent of the nodes representing the sets of states with a partial signature $a_1p_1a_2p_2\cdots a_ip_ia_{i+1}p_{i+1}$. As a consequence, leaves represent full signatures. All states that correspond to a leaf are mergeable.

When mergeable states are detected in the signature tree, they can be merged. Then the signature tree has to be updated, and this is the difficult part of the algorithm.

7.2 Bottom-up minimization

In this section, all automata are finite, acyclic, deterministic and trim. A state p is called *confluent* if there are at least two edges in $\mathcal{A}$ ending in p.

A *trie* is an automaton whose underlying graph is a tree. Thus an automaton is a trie if and only if it has no confluent state.

Bottom-up minimization is the process of minimizing an acyclic automaton by a bottom-up traversal. In such a traversal, children of a node are treated before the node itself. During the traversal, equivalent states are detected and merged. The basic property of bottom-up minimization is that the check for (Nerode) equivalence reduces to equality of signatures. The critical point is to organize the states that are candidates in order to do this check efficiently.

The bottom-up traversal itself may be organized in several ways, for instance as a depth-first search with the order of traversal of the children determined by the order on the labels of the edges. Another traversal is by increasing height, as done in Revuz's algorithm given next.

```
REVUZ(𝒜)
  for h = 0 to HEIGHT(𝒜) do
    S ← GETSTATESFORHEIGHT(h)              ▷ Compute states of height h
    SORTSIGNATURES(S)                      ▷ Compute and sort signatures
    for s ∈ S do                           ▷ Merge mergeable states
      if s and s.next have the same signature then
        MERGE(s, s.next)                   ▷ Mergeable states are consecutive
```

Figure 10. Revuz's minimization algorithm.

One popular method for the construction of a minimal automaton for a given finite set of words consists in first building a trie for this set and then minimizing it. Daciuk *et al.* [27] propose an incremental version which avoids this intermediate construction.

Recall that the *signature* of a state p is the set of pairs (a, q) such that $p \cdot a = q$ together with a Boolean value indicating whether the state is final or not. It is tacitly understood that the alphabet of the automaton is ordered. The signature of a state is usually considered as the ordered sequence of pairs, where the order is determined by the letters. It is important to observe that the signature of a state evolves when states are merged. As an example, the state 6 of the automaton on the left of Figure 11 has signature $+, (a, 3), (b, 10)$, and the same state has signature $+, (a, 3), (b, 7)$ in the automaton on the right of the figure.

As already mentioned, if the minimization of the children of two states p and q has been done, then p and q are (Nerode) equivalent if and only if they have the same signature. So the problem to be considered is the bookkeeping of signatures, that is the problem of detecting whether the signature of the currently considered state has already occured before. In practical implementations, this is done by hash coding the signatures. This allows one to perform the test in constant average time. One remarkable exception is Revuz's algorithm to be presented now, and its extension by Almeida and Zeitoun that we describe later.

7.3 Revuz's algorithm

Revuz [47] was the first to give an explicit description of a linear time implementation of the bottom-up minimization algorithm. The principle of the algorithm was also described by [41].

Define the *height* of a state p in an acyclic automaton to be the length of the longest path starting at p. It is also the length of the longest word in the language of the subautomaton at p. Two equivalent states have the same height. Revuz's algorithm operates by increasing height. It is outlined in Figure 10. Heights may be computed in linear time by a bottom-up traversal. The lists of states of a given height are collected during this traversal. The signature of a state is easy to compute provided the edges starting in a state have been sorted (by a bucket sort for instance to remain within the linear time constraint). Sorting states by their signature again is done by a lexicographic sort. As for Moore's algorithm, the last step can by done by a simple scan of the list of states since states with equal signature are consecutive in the sorted list.

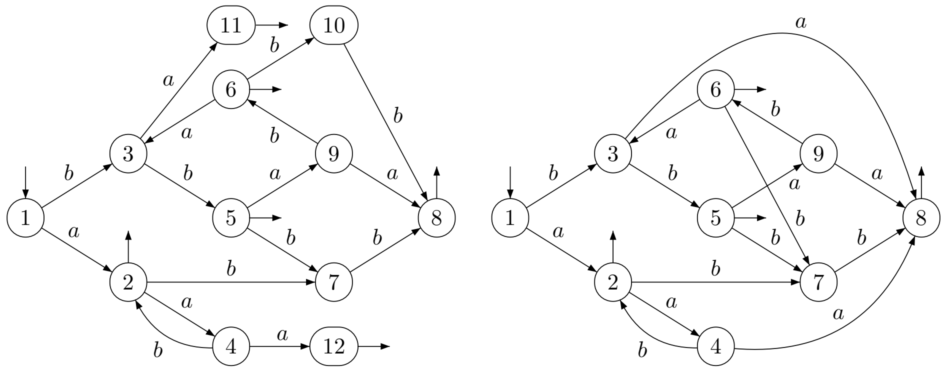

Figure 11. On the left a simple automaton: its nontrivial strongly connected components are the cycles $2, 4$ and $3, 5, 9, 6$. The minimization starts by merging 11, 12 and 8, and also 10 and 7. This gives the automaton on the right.

The whole algorithm can be implemented to run in time $O(m)$ for an automaton with m edges.

Revuz's algorithm relies on a particular bottom-up traversal of the trie. This traversal is defined by increasing height of states, and it makes the check for equivalent states easier. With another method for checking signatures in mind, like hash coding, the algorithm may be organized in a different way. For instance, the traversal by heights can be replaced by a traversal by lexicographic order. The last item in the algorithm may be replaced by another check. Whenever a state has been found which must be in the minimal automaton, its hash code is *registered*. When the signature of a state is computed, one checks whether its hash code is registered. If not, it is added to the register, otherwise it is replaced by the hash code.

Several implementations have been given in various packages. A comparison has been given in [25]. See also Table 2 below for numerical data.

7.4 The algorithm of Almeida and Zeitoun

Almeida and Zeitoun [3] consider an extension of the bottom-up minimization algorithm to automata which contain only simple cycles. They describe a linear time algorithm for these automata.

Let $\mathcal{A}$ be a finite trim automaton. We call it *simple* if every nontrivial strongly connected component is a simple cycle, that is if every vertex of the component has exactly one successor vertex in this component. The automaton given on the left of Figure 11 is simple. Simple automata are interesting because they recognize exactly the bounded regular languages or, equivalently, the languages with polynomial growth. These are the simplest infinite regular languages.

The starting point of the investigation of [3] is the observation that minimization can be split into two parts: minimization of an acyclic automaton and minimization of the set of strongly connected components. There are three subproblems, namely (1) minimization of each strongly connected component, (2) identification and fusion of isomorphic minimized strongly connected components, and (3) wrapping, which consists in merging states which are equivalent to a state in a strongly connected component, but which are not in this component. The authors show that if these subproblems can be solved in linear time, then, by a bottom-up algorithm which is a sophistication of Revuz's algorithm, the whole automaton can be minimized in linear time. Almeida and Zeitoun show how this can be done for simple automata. The outline of the algorithm is given in Figure 12.

The algorithm works as Revuz's algorithm as long as no nontrivial strongly connected components occur. In our example automaton, the states 8, 11 and 12 are merged, and the states 10 and 7 also are merged. This gives the automaton on the right of Figure 11.

Then a cycle which has all its descendants minimized is checked for possible minimization. This is done as follows: the *weak signature* of a state p of a cycle is the signature obtained by replacing the name of its successor in the cycle by a dummy symbol, say $\square$. In our example, the weak signatures of the states $3, 5, 9, 6$ are respectively:

$$-a8b\square\,, \quad +a\square b7\,, \quad -a8b\square\,, \quad +a\square b7\,.$$

Here we write '$+$' when the state is final, and '$-$' otherwise.

It is easily seen that the cycle is minimal if and only if the word composed of the sequence of signatures is primitive. In our example, the word is not primitive since it is a square, and the cycle can be reduced by identifying states that are at corresponding positions in the word, that is states 5 and 6 can be merged, and states 3 and 9. This gives the automaton on the left of Figure 13.

Similarly, in order to check whether two (primitive) cycles can be merged, one checks whether the words of their weak signatures are conjugate. In our example, the cycles $2, 4$ and $3, 5$ have the signatures

$$+a\square b7\,, -a8b\square \quad \text{and} \quad -a8b\square\,, +a\square b7\,.$$

These words are conjugate and the corresponding states can be merged. This gives the automaton on the right of Figure 13. This automaton is minimal.

A basic argument for preserving the linearity of the algorithm is the fact that the minimal conjugate of a word can be computed in linear time. This can be done for instance by Booth's algorithm (see [24]). Thus, testing whether a cycle is minimized takes time proportional to its length, and for each minimized cycle, a canonical representative, namely the unique minimal conjugate, which is a Lyndon word, can be computed in time proportional to its length. The equality of two cycles then reduces to the equality of the two

```
AlmeidaZeitoun(A)
  S ← ZeroHeight(A)                  ▷ States and cycles of height 0
  while S ≠ ∅ do
    MinimizeCycles(S)                ▷ Minimize each cycle
    MergeIsomorphicCycles(S)         ▷ Compute and sort signatures and merge
    Wrap(S)                          ▷ Search states to wrap
    S ← NextHeight(A, S)             ▷ Compute states for next height
```

Figure 12. The algorithm of Almeida and Zeitoun.

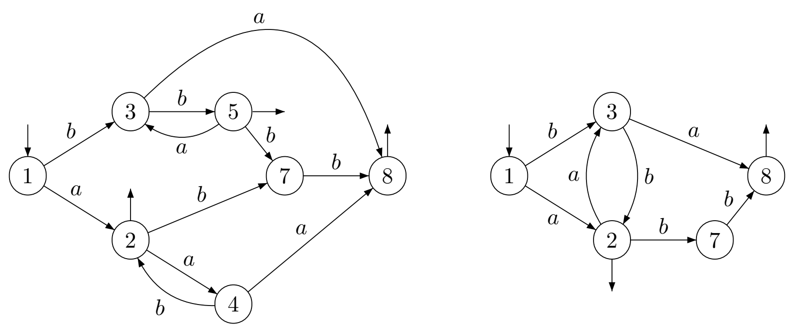

Figure 13. The minimization continues by merging the states 5 and 6, and the states 3 and 9. This gives the automaton on the left. The last step of minimization merges the states 2 and 5, and the states 3 and 4.

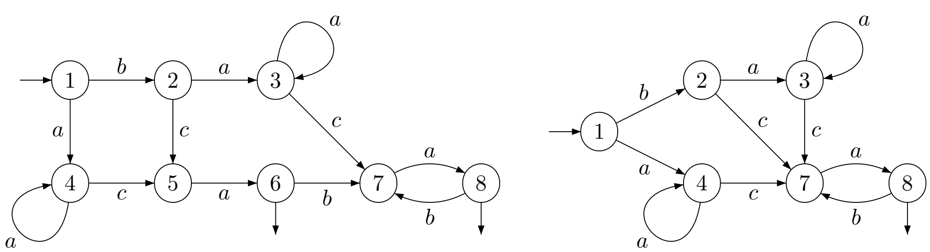

Figure 14. The automaton on the left has one minimal cycle of height 1. By wrapping, states 6 and 8, and states 5 and 7 are merged, respectively, giving the automaton on the right.

associated words. Finding isomorphic cycles is accomplished by a lexicographic ordering of the associated words, followed by a simple scan for equal words.

A few words on wrapping: it may happen that states of distinct heights in a simple automaton are equivalent. An example is given in Figure 14. Indeed, states 6 and 8 have the same signature and therefore are mergeable but have height 1 and 0, respectively. This situation is typical: when states s and t are mergeable and have distinct heights, and t belongs to a minimized component of current height, then s is a singleton component on a path to the cycle of t. Wrapping consists in detecting these states, and in "winding" them around the cycle. In our example, both 6 and 5 are wrapped in the component of 7 and 8. In the algorithm given above, a wrapping step is performed at each iteration, after the minimization of the states and the cycles and before computing the states and cycles of the next height. In our example, after the first iteration, states 3 and 4 are mergeable. A second wrapping step merges 3 and 4. These operations are reported in Figure 15. A careful implementation can realize all these operations in global linear time.

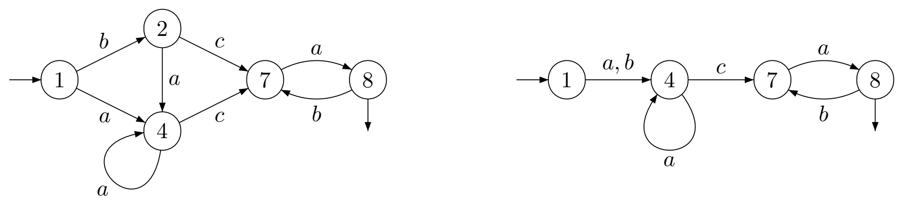

Figure 15. The automaton on the left has one minimal cycle of height 1. By wrapping, states 6 and 8, and states 5 and 7 are merged, respectively, giving the automaton on the right.

7.5 Incremental minimization: the algorithm of Daciuk *et al.*

The algorithm presented in [27] is an incremental algorithm for the construction of a minimal automaton for a given set of words that is lexicographically sorted.

The algorithm is easy to implement and it is efficient: the construction of an automaton recognizing a typical dictionary is done in a few seconds. Table 2 was kindly communicated by Sébastien Paumier. It contains the space saving and the computation time for dictionaries of various languages.

The algorithm described here is simple because the words are sorted. There exist other incremental algorithms for the case of unsorted sets. One of them will be described in the next section. Another algorithm, called semi-incremental because it requires a final minimization step, is given in [51].

We start with some notation. Let $\mathcal{A} = (Q, i, F)$ be a finite, acyclic, deterministic and trim automaton. We say that a word x *is in* the automaton $\mathcal{A}$ if $i \cdot x$ is defined. In other words, x is in $\mathcal{A}$ if x is a prefix of some word recognized by $\mathcal{A}$. Let w be a word to be added to the set recognized by an automaton $\mathcal{A}$. The factorization

$$w = x \cdot y\,,$$

where x is the longest prefix of w which is in $\mathcal{A}$, is called the *prefix-suffix decomposition* of w. The word x (resp. y) is the *common prefix* (resp. *corresponding suffix*) of w.

One has $x = \varepsilon$ if either $w = \varepsilon$ or $i \cdot a$ is undefined, where a is the initial letter of w. Similarly, $y = \varepsilon$ if w itself is in $\mathcal{A}$. If $y \neq \varepsilon$ and starts with the letter b, then $i \cdot xb = \perp$.

The *insertion* of a word y at state p is an operation that is performed provided $y = \varepsilon$ or $p \cdot b = \perp$, where b is the initial letter of y. If $y = \varepsilon$, the insertion simply consists in adding state p to the set F of final states. If $y \neq \varepsilon$, set $y = b_1 b_2 \cdots b_m$. The insertion consists in adding new states $p_1, \ldots, p_m$ to Q, with the next state function defined by $p \cdot b_1 = p_1$ and $p_{i-1} \cdot b_i = p_i$ for $i = 2, \ldots, m$. Furthermore, p_m is added to the set F of final states.

Assume that the language recognized by $\mathcal{A}$ is not empty, and that the word w is lexicographically greater than all words in $\mathcal{A}$. Then w is not in $\mathcal{A}$. So the common prefix x of w is strictly shorter than w and the corresponding suffix y is nonempty.

The incremental algorithm works at follows. At each step, a new word w that is lexicographically greater than all previous ones is inserted in the current automaton $\mathcal{A}$. First, the prefix-suffix decomposition $w = xy$ of w, and the state $q = i \cdot x$ are computed. Then the segment starting at q of the path carrying the suffix y' of the previously inserted

File	Lines	Text file	Automaton			Time	
			States	Trans.	Size	Revuz	Daciuk
delaf-de	189878	12.5Mb	57165	103362	1.45Mb	4.22s	4.44s
delaf-en	296637	13.2Mb	109965	224268	2.86Mb	5.94s	6.77s
delaf-es	638785	35.4Mb	56717	117417	1.82Mb	10.61s	11.28s
delaf-fi	256787	24.6Mb	124843	133288	4.14Mb	6.40s	7.02s
delaf-fr	687645	38.7Mb	109466	240409	3.32Mb	13.03s	14.14s
delaf-gr	1288218	83.8Mb	228405	442977	7.83Mb	28.33s	31.02s
delaf-it	611987	35.9Mb	64581	161718	1.95Mb	10.43s	11.46s
delaf-no	366367	23.3Mb	75104	166387	2.15Mb	6.86s	7.44s
delaf-pl	59468	3.8Mb	14128	20726	502Kb	1.19s	1.30s
delaf-pt	454241	24.8Mb	47440	115694	1.4Mb	7.87s	8.45s
delaf-ru	152565	10.8Mb	23867	35966	926Kb	2.95s	3.17s
delaf-th	33551	851Kb	36123	61357	925Kb	0.93s	1.14s

Table 2. Running time and space requirement for the computation of minimal automata (*communication of Sébastien Paumier*).

```
DaciukEtAl(A)
  for all w do                          ▷ Words are given in lexicographic order
    (x, y) ← PrefSuffDecomp(w)          ▷ x is the longest prefix of w in A
    q ← i · x                           ▷ q is the state reached by reading x
    MinimizeLastPath(q)                 ▷ Minimize the states on this path
    AddPath(q, y)                       ▷ Adds a path starting in q and carrying y
```

Figure 16. The incremental algorithm of Daciuk *et al*.

word w' is minimized by merging states with the same signature. Finally, the suffix y is inserted at state q. The algorithm is given in Figure 16.

The second step deserves a more detailed description. We observe first that the word x of the prefix-suffix decomposition $w = xy$ of w is in fact the greatest common prefix of w' and w. Indeed, the word x is a prefix of some word recognized by $\mathcal{A}$ (here $\mathcal{A}$ is the automaton before adding w), and since w' is the greatest word in $\mathcal{A}$, the word x is a prefix of w'. Thus x is a common prefix of w' and w. Next, if x' is a common prefix of w' and w, then x' is in $\mathcal{A}$ because it is a prefix of w', and consequently x' is a prefix of x because x is the longest prefix of w in $\mathcal{A}$. This shows the claim.

There are two cases for the merge. If w' is a prefix of w, then $w' = x$. In this case, there is no minimization to be performed.

If w' is not a prefix of w, then the paths for w' and for w share a common initial segment carrying the prefix x, from the initial state to state $q = i \cdot x$. The minimization concerns the states on the path $q \xrightarrow{y'} t'$ carrying the suffix y' of the factorization $w = xy'$ of w'. Each of the states in this path, except the state q, will never be visited again in any insertion that may follow, so they can be merged with previous states.

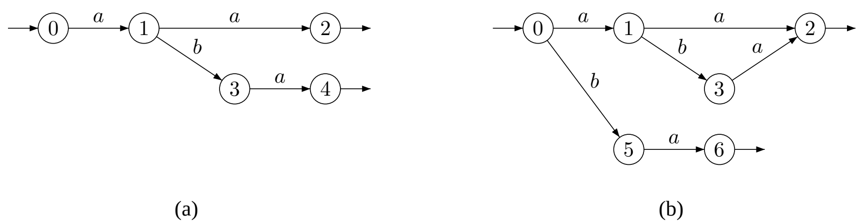

Figure 17. (a). The automaton for aa, aba. (b). The automaton for aa, aba, ba. Here state 4 has been merged with state 2.

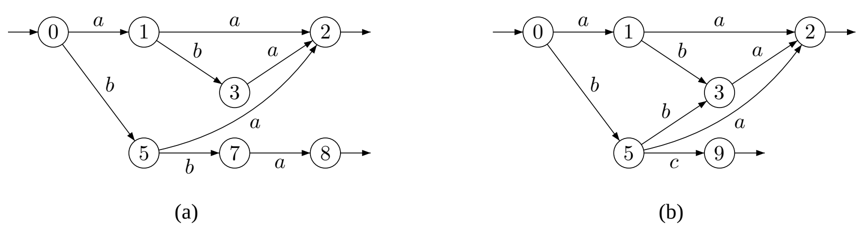

Figure 18. (a). The automaton for aa, aba, ba, aba, bba. (b). The automaton for $aa, aba, ba, aba, bba, bc$. After inserting bc, states 8 and 2 are merged, and then states 7 and 2.

Example 7.1. We consider the sequence of words (aa, aba, ba, bba, bc). The first two words give the automaton of Figure 17(a). Adding the word ba permits the merge of states 2 and 4. The resulting automaton is given in Figure 17(b). After inserting bba, there is a merge of states 6 and 2, see Figure 18(a).

8 Dynamic minimization

Dynamic minimization is the process of maintaining an automaton minimal when insertions or deletions are performed.

A solution for adding and for removing a word was proposed by Carrasco and Forcada [19]. It consists in an adaptation of the usual textbook constructions for intersection and complement to the special case where one of the languages is a single word. It appears that the finiteness of the language L plays no special role, so we assume here that it is regular, not necessarily finite. The construction for adding a word has also been proposed in [49], and in [27] for acyclic automata. An extension to general automata, and several other issues, are discussed in [26].

We consider here, for lack of space, only deletion of a word from the set recognized by an automaton, and minimization of the new automaton.

Let $\mathcal{A} = (Q, i, T)$ be the minimal automaton recognizing a language L over the alphabet A, and let w be a word in L. Denote by $\mathcal{A}_w$ the minimal automaton recognizing

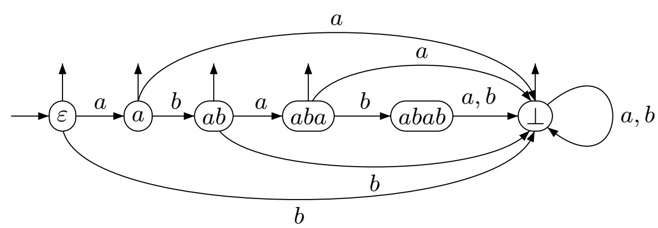

Figure 19. The minimal automaton recognizing the complement of the word $abab$. Only the state $abab$ is not final.

the complement $A^* \setminus w$. The automaton has $n+2$ states, with $n = |w|$. Among them, there are $n+1$ states that are identified with the set P of prefixes of w. The last state is a sink state denoted $\perp$. An example is given in Figure 19.

The language $L \setminus w$ is equal to $L \cap (A^* \setminus w)$, so it is recognized by the trimmed part B of the product automaton $\mathcal{A} \times \mathcal{A}_w$. Its initial state is (i, ε), and its states are of three kinds.

- *intact states*: these are states of the form $(q, \perp)$ with $q \in Q$. They are called so because the language recognized at $(q, \perp)$ in $\mathcal{B}$ is the same as the language recognized at q in $\mathcal{A}$: $L_{\mathcal{B}}(q, \perp) = L_{\mathcal{A}}(q)$.
- *cloned states*: these are accessible states (q, x) with $x \in P$, so x is a prefix of w. Since we require these states to be accessible, one has $q = i \cdot x$ in $\mathcal{A}$, and there is one such state for each prefix. The next-state function on these states is defined by

$$(q, x) \cdot a = \begin{cases} (q \cdot a, xa) & \text{if } xa \in P, \\ (q \cdot a, \perp) & \text{otherwise.} \end{cases}$$

  Observe that $(i \cdot w, \perp)$ is an intact state because w is assumed to be recognized by $\mathcal{A}$.
- *useless states*: these are all states that are removed when trimming the product automaton.

Trimming consists here in removing the state $(i, \perp)$ if it is no longer accessible, and the states reachable only from this state. For this, one follows the path defined by w and starting in $(i, \perp)$ and removes the states until one reaches a confluent state (that has at least two incoming edges). The automaton obtained is minimal.

The whole construction finally consists in keeping the initial automaton, by renaming a state q as $(q, \perp)$, adding a cloned path, and removing state $(i, \perp)$ if it is no longer accessible, and the states reachable only from this state.

Of course, one may also use the textbook construction directly, that is without taking advantage of the existence of the automaton given at the beginning. For this, one starts at the new initial state (i, ε) and one builds only the accessible part of the product automaton. The method has complexity $O(n + |w|)$, where n is the number of states of the initial automaton, whereas the previous method has only complexity $O(|w|)$.

Example 8.1. The automaton given in Figure 21 recognizes the language $L = (ab)^+ \cup \{abc, acb\}$. The direct product with the automaton of Figure 19 is shown in Figure 22. Observe that there are intact states that are not accessible from the new initial state $(0, \varepsilon)$.

REMOVEINCREMENTAL$(w, \mathcal{A})$
  $\mathcal{A}' \leftarrow$ ADDCLONEDPATH$(w, \mathcal{A})$ ▷ Add a fresh path for w in $\mathcal{A}$
  TRIM$(\mathcal{A}')$ ▷ Return trimmed automaton

ADDCLONEDPATH$(a_1 \cdots a_n, \mathcal{A})$
  $p_0 \leftarrow$ INITIAL$(\mathcal{A})$; $q_0 \leftarrow$ CLONE(p_0) ▷ Add a fresh initial state q_0
  for $i = 1$ **to** n **do**
    $p_i \leftarrow p_{i-1} \cdot a_i$; $q_i \leftarrow$ CLONE(p_i) ▷ q_i inherits the transitions of p_i
    $q_{i-1} \cdot a_i \leftarrow q_i$ ▷ This edge is redirected
  SETFINAL$(q_n,$ **false**$)$

Figure 20. Removing the word w from the language recognized by $\mathcal{A}$.

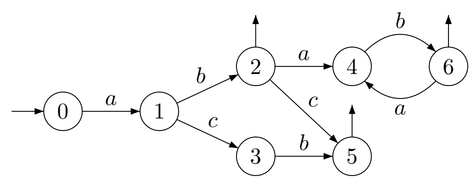

Figure 21. The minimal automaton recognizing the language $L = (ab)^+ \cup \{abc, acb\}$.

The minimal automaton is shown in Figure 23.

9 Extensions and special cases

In this section, we consider extensions of the minimization problem to other classes of automata. The most important problem is to find a minimal nondeterministic automaton

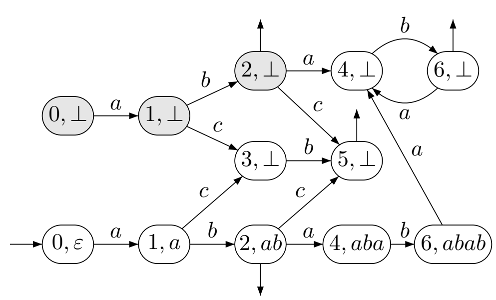

Figure 22. The automaton recognizing the language $L = (ab)^+ \cup \{abc, acb\} \setminus \{abab\}$. There are still unreachable states (shown in gray).

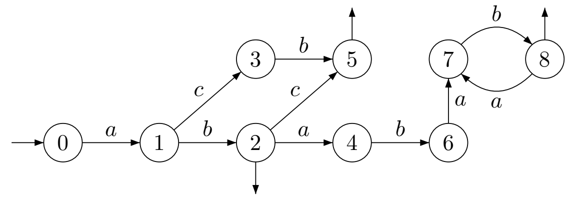

Figure 23. The minimal automaton for the language $L = (ab)^+ \cup \{abc, acb\} \setminus \{abab\}$.

recognizing a given regular language. Other problems, not considered here, concern sets of infinite words and the minimization of their accepting devices, and the use of other kinds of automata known to be equivalent with respect to their accepting capability, such as two-way automata or alternating automata, see for instance [46].

9.1 Special automata

We briefly mention here special cases where minimization plays a role. It is well known that *string matching* is closely related to the construction of particular automata. If w is a nonempty word over an alphabet A, then searching for all occurrences of w as a factor in a text t is equivalent to computing all prefixes of t ending in w, and hence to determining all prefixes of t which are in the regular language A^*w. The minimal automaton recognizing A^*w has $n+1$ states, where $n = |w|$, and can be constructed in linear time. The automaton has many interesting properties. For instance there are at most $2n$ edges, when one does not count edges ending in the initial state. This is due to Imre Simon, see also [33]. For a general exposition, see e.g. [24]. Extension of string matching to a finite set X of patterns has been done by Aho and Corasick. The associated automaton is called the *pattern matching machine*; it can be computed in time linear in the sum of the lengths of the words in X. See again [24]. However, this automaton is not minimal in general. Indeed, the number of states is the number of distinct prefixes of words in X, and this may be greater than than the number of states in the minimal automaton (consider for example the set $X = \{ab, bb\}$ over the alphabet $A = \{a, b\}$). There are some investigations on the complexity of minimizing Aho–Corasick automata, see [2].

Another famous minimal automaton is the *suffix automaton*. This is the minimal automaton recognizing all suffixes of a given word. The number of states of the suffix automaton of a word of length n is less than $2n$, and the number of its edges is less than $3n$. Algorithms for constructing suffix automata in linear time have been given in [23] and [17], see again [24] for details.

9.2 Nondeterministic automata

A nondeterministic automaton is minimal if it has the minimal number of states among all automata recognizing the same language. Nondeterministic automata are not unique. In Figure 24, we give two non-isomorphic nondeterministic automata which are both smaller

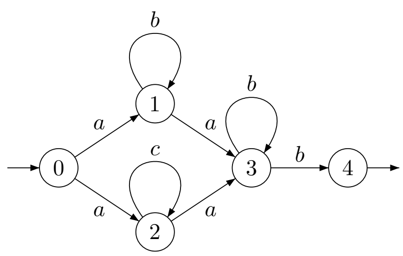

Figure 24. Two non-isomorphic non-deterministic automata recognizing the set $a(b^* \cup c^*)ab^+$.

than the minimal deterministic automaton recognizing the same language. This language is $a(b^* \cup c^*)ab^+$. The example is derived from an automaton given in [4].

One might ask if there are simple conditions on the automata or on the language that ensure that the minimal nondeterministic automaton is unique. For instance, the automata of Figure 25 both recognize the same language, but the second has a particular property that we will describe now. The uniqueness of the minimal automaton in the deterministic

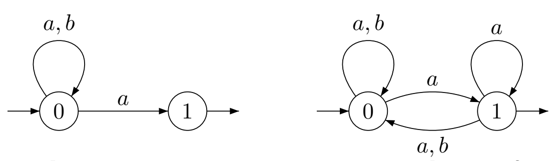

Figure 25. Two nondeterministic automata recognizing the set of words ending with the letter a.

case is related to the fact that the futures of the states of such an automaton are pairwise distinct, and that each future is some left quotient of the language: for each state q, the language $L_q(\mathcal{A})$ is equal to a set $y^{-1}L$, for some word y.

This characterization has been the starting point for investigating similar properties of nondeterministic automata. Let us call a (nondeterministic) automaton a *residual automaton* if the future of its states are left quotients of the language; it has been shown in [30] that, among all residual automata recognizing a given language, there is a unique residual automaton having a minimal number of states; moreover, this automaton is characterized by the fact that the set of its futures is the set of the prime left quotients of the language, a left quotient being *prime* if it is not the union of other nonempty left quotients. For instance, the automaton on the right of Figure 25 has this property, since $L_0 = \{a, b\}^*a$ and $L_1 = a^{-1}L_0 = \varepsilon \cup \{a, b\}^*a$ and there are no other nonempty left quotients. The automaton on the left of Figure 25 is not residual since the future of state 1 is not a left quotient.

The problem of converting a given nondeterministic automaton into a minimal nondeterministic automaton is NP-hard, even over a unary alphabet [36]. This applies also to unambiguous automata [43]. In [15], these results have been extended as follows. The authors define a class δNFA of automata that are unambiguous, have at most two computations for each string, and have at most one state with two outgoing transitions carrying the same letter. They show that minimization is NP-hard for all classes of finite automata that include δNFA, and they show that these hardness results can also be adapted to the setting of unambiguous automata that can non-deterministically choose between two start states, but are deterministic everywhere else.

Even approximating minimization of nondeterministic automata is intractable, see [31]. There is an algebraic framework that allows one to represent and to compute all automata recognizing a given regular language. The state of this theory, that goes back to Kameda and Weiner [39], has been described by Lombardy and Sakarovitch in a recent survey paper [42].

There is a well-known exponential blow-up from nondeterministic automata to deterministic ones. The usual textbook example, already given in the first section (the automaton on the left in Figure 1) shows that this blow-up holds also for unambiguous automata, even if there is only one edge that causes the nondeterminism.

It has been shown that any value of blow-up can be obtained, in the following sense [37]: for all integers n, N with $n \leqslant N \leqslant 2^n$, there exists a minimal nondeterministic automaton with n states over a four-letter alphabet whose equivalent minimal deterministic automaton has exactly N states. This was improved to ternary alphabets [38].

Acknowledgements

We had several helpful discussions with Marie-Pierre Béal, Julien David, Sylvain Lombardy, Wim Martens, Cyril Nicaud, Sébastien Paumier, Jean-Éric Pin and Jacques Sakarovitch. We thank Narad Rampersad for his careful reading of the text.

References

[1] A. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974. 8

[2] O. AitMous, F. Bassino, and C. Nicaud. Building the minimal automaton of A^*X in linear time, when X is of bounded cardinality. In A. Amir and L. Parida, editors, *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010*, volume 6129 of *Lecture Notes in Comput. Sci.*, pages 275–287. Springer-Verlag, 2010. 29

[3] J. Almeida and M. Zeitoun. Description and analysis of a bottom-up DFA minimization algorithm. *Inform. Process. Lett.*, 107(2):52–59, 2008. 2, 21

[4] A. Arnold, A. Dicky, and M. Nivat. A note about minimal non-deterministic automata. *Bull. European Assoc. Theor. Comput. Sci.*, 47:166–169, 1992. 30

[5] M. Baclet and C. Pagetti. Around Hopcroft's algorithm. In *11th Conference on Implementation Application Automata (CIAA)*, volume 4094 of *Lecture Notes in Comput. Sci.*, pages 114–125. Springer-Verlag, 2006. 12

[6] F. Bassino, J. David, and C. Nicaud. Average case analysis of Moore's state minimization algorithm. *submitted*. 8

[7] F. Bassino, J. David, and C. Nicaud. On the average complexity of Moore's state minimization algorithm. In S. Albers and J.-Y. Marion, editors, *STACS 2009, Proc. 26th Symp. Theoretical Aspects of Comp. Sci.*, volume 09001 of *Dagstuhl Seminar Proceedings*, pages 123–134. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2009. 2, 8

[8] M.-P. Béal and M. Crochemore. Minimizing local automata. In G. Caire and M. Fossorier, editors, *IEEE International Symposium on Information Theory (ISIT'07)*, number 07CH37924C, pages 1376–1380. IEEE, 2007. 2, 18

[9] M.-P. Béal and M. Crochemore. Minimizing incomplete automata. In *Workshop on Finite State Methods and Natural Language Processing 2008*. Ispra, september 2008. 2, 13

[10] D. Beauquier, J. Berstel, and P. Chrétienne. *Éléments d'algorithmique*. Masson, 1992. 8

[11] J. Berstel, L. Boasson, and O. Carton. Continuant polynomials and worst-case behavior of Hopcroft's minimization algorithm. *Theoret. Comput. Sci.*, 410(30-32):2811–2822, 2009. 17

[12] J. Berstel, L. Boasson, O. Carton, and I. Fagnot. Sturmian trees. *Theory Comput. Systems*, 46:443–478, 2010. 17

[13] J. Berstel and O. Carton. On the complexity of Hopcroft's state minimization algorithm. In *9th Conference on Implementation Application Automata (CIAA)*, volume 3317 of *Lecture Notes in Comput. Sci.*, pages 35–44. Springer-Verlag, 2004. 12, 17

[14] J. Berstel and D. Perrin. Algorithms on words. In M. Lothaire, editor, *Algebraic combinatorics on words*, volume 90 of *Encyclopedia of Mathematics and its Applications*, chapter 1, pages 1–100. Cambridge University Press, 2002. 12

[15] H. Björklund and W. Martens. The tractability frontier for NFA minimization. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008*, volume 5126 of *Lecture Notes in Comput. Sci.*, pages 27–38. Springer-Verlag, 2008. 30

[16] N. Blum. A $O(n \log n)$ implementation of the standard method for minimizing n-state finite automata. *Inform. Process. Lett.*, 57:65–69, 1996. 8

[17] A. Blumer, J. A. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoret. Comput. Sci.*, 40:31–55, 1985. 29

[18] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Proc. Sympos. Math. Theory of Automata (New York, 1962)*, pages 529–561. Polytechnic Press of Polytechnic Inst. of Brooklyn, Brooklyn, N.Y., 1963. 5

[19] R. C. Carrasco and M. L. Forcada. Incremental construction and maintenance of minimal finite-state automata. *Comput. Linguist.*, 28(2), 2002. 26

[20] G. Castiglione, A. Restivo, and M. Sciortino. Circular words and automata minimization. In P. Arnoux, N. Bédaride, and J. Cassaigne, editors, *Words 2007*, pages 79–89. Institut de Mathématiques de Luminy, 17–21 september 2007. 17

[21] G. Castiglione, A. Restivo, and M. Sciortino. Hopcroft's algorithm and cyclic automata. In C. Martín-Vide, F. Otto, and H. Fernau, editors, *Language and Automata Theory and Applications*, volume 5196 of *Lecture Notes in Comput. Sci.*, pages 172–183. Springer-Verlag, 2008. Tarragona, Spain, March 13-19, 2008. Revised Papers. 17

[22] G. Castiglione, A. Restivo, and M. Sciortino. Circular Sturmian words and Hopcroft's algorithm. *Theoret. Comput. Sci.*, 410:4372–4381, 2009. 17

[23] M. Crochemore. Transducers and repetitions. *Theoret. Comput. Sci.*, 45:63–86, 1986. 29

[24] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings*. Cambridge University Press, 2007. 22, 29

[25] J. Daciuk. Comparison of construction algorithms for minimal, acyclic, deterministic finite-state automata from sets of strings. In J.-M. Champarnaud and D. Maurel, editors, *7th Conference on Implementation Application Automata (CIAA)*, volume 2608 of *Lecture Notes in Comput. Sci.*, pages 255–261. Springer-Verlag, 2002. 21

[26] J. Daciuk. Comments on ”Incremental construction and maintenance of minimal finite-state automata” by Rafael C. Carrasco and Mikel L. Forcada. *Comput. Linguist.*, 30(2):227–235, 2004. 26

[27] J. Daciuk, S. Mihov, B. W. Watson, and R. E. Watson. Incremental construction of minimal acyclic finite-state automata. *Comput. Linguist.*, 26(1):3–16, april 2000. 3, 20, 24, 26

[28] J. David. The average complexity of Moore's state minimization algorithm is $o(n \log \log n)$. In P. Hlinený and A. Kucera, editors, *Mathematical Foundations of Computer Science 2010, 35th International Symposium, MFCS 2010*, volume 6281 of *Lecture Notes in Comput. Sci.*, pages 318–329. Springer-Verlag, 2010. 8

[29] J. David. *Génération aléatoire d'automates et analyse d'algorithmes de minimisation*. Doctorat d'unversité, University Paris-Est, september 2010. 14

[30] F. Denis, A. Lemay, and A. Terlutte. Residual finite state automata. In *STACS 2001, Proc. 18th Symp. Theoretical Aspects of Comp. Sci.*, volume 2010 of *Lecture Notes in Comput. Sci.*, pages 144–157. Springer-Verlag, 2001. 30

[31] G. Gramlich and G. Schnitger. Minimizing NFA's and regular expressions. In *STACS 2005, Proc. 22th Symp. Theoretical Aspects of Comp. Sci.*, volume 3404 of *Lecture Notes in Comput. Sci.*, pages 399–411. Springer-Verlag, 2005. 31

[32] D. Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973. 8

[33] C. Hancart. On Simon's string searching algorithm. *Inform. Process. Lett.*, 47(2):95–99, 1993. 29

[34] J. E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971. 2, 5, 8, 12

[35] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley Publishing Co., Reading, Mass., 1979. Addison-Wesley Series in Computer Science. 2

[36] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM J. Comput.*, 22(6):1117–1141, 1993. 30

[37] J. Jirásek, G. Jirásková, and A. Szabari. Deterministic blow-up of minimal nondeterministic finite automata over a fixed alphabet. In J. Karhumäki and A. Lepistö, editors, *Developments in Language Theory*, volume 4588 of *Lecture Notes in Comput. Sci.*, pages 254–265. Springer-Verlag, 2007. 31

[38] G. Jirásková. Magic numbers and ternary alphabet. In V. Diekert and D. Nowotka, editors, *Developments in Language Theory*, volume 5583 of *Lecture Notes in Comput. Sci.*, pages 300–311. Springer-Verlag, 2009. 31

[39] T. Kameda and P. Weiner. On the state minimization of nondeterministic finite automata. *IEEE Trans. Comput.*, C-19(7):617–627, 1970. 31

[40] T. Knuutila. Re-describing an algorithm by Hopcroft. *Theoret. Comput. Sci.*, 250:333–363, 2001. 8, 12

[41] S. L. Krivol. Algorithms for minimization of finite acyclic automata and pattern matching in terms. *Cybernetics*, 27:324– 331, 1991. translated from Kibernetika, No 3, May-June 1991, pp. 11–16. 20

[42] S. Lombardy and J. Sakarovitch. The universal automaton. In E. Grädel, J. Flum, and T. Wilke, editors, *Logic and automata: History and perspectives*, volume 2 of *Texts in Logic and Games*, pages 467–514. Amsterdam University Press, 2007. 31

[43] A. Malcher. Minimizing finite automata is computationally hard. *Theoret. Comput. Sci.*, 327(3):375–390, 2004. 30

[44] E. F. Moore. Gedanken experiments on sequential machines. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 129–153. Princeton Universty Press, 1956. 2, 6

[45] A. Paŭn, M. Paŭn, and A. Rodríguez-Patón. On Hopcroft's minimization technique for DFA and DFCA. *Theoret. Comput. Sci.*, 410:2424–2430, 2009. 12

[46] D. Perrin and J.-E. Pin. *Infinite words, automata, semigroups, logic and games*. Elsevier, 2004. 29

[47] D. Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoret. Comput. Sci.*, 92:181–189, 1992. 2, 20

[48] J. Sakarovitch. *Elements of automata theory*. Cambridge University Press, 2009. 6

[49] K. N. Sgarbas, N. D. Fakotakis, and G. K. Kokkinakis. Optimal insertion in deterministic DAWGs. *Theoret. Comput. Sci.*, 301:103–117, 2003. 26

[50] A. Valmari and P. Lehtinen. Efficient minimization of DFAs with partial transition. In S. Albers and P. Weil, editors, *STACS 2008, Proc. 25th Symp. Theoretical Aspects of Comp. Sci.*, volume 08001 of *Dagstuhl Seminar Proceedings*, pages 645–656. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008. 2, 13

[51] B. W. Watson. A new algorithm for the construction of minimal acyclic DFAs. *Sci. Comput. Program.*, 48(2-3):81–97, 2003. 24

Abstract. This chapter is concerned with the design and analysis of algorithms for minimizing finite automata. Getting a minimal automaton is a fundamental issue in the use and implementation of finite automata tools in frameworks like text processing, image analysis, linguistic computer science, and many other applications.

Although minimization algorithms are rather old, there were recently some new developments which are explained or sketched in this chapter.

There are two main families of minimization algorithms. The first are by a sequence of refinements of a partition of the set of states, the second by a sequence of fusions or merges of states. Hopcroft's and Moore's algorithms belong to the first family. The linear-time minimization of acyclic automata of Revuz belongs to the second family.

We describe, prove and analyze Moore's and Hopcroft's algorithms. One of our studies is the comparison of these algorithms. It appears that they quite different both in behavior and in complexity. In particular, we show that it is not possible to simulate the computations of one of the algorithm by the other. We also state the results known about average complexity of both algorithms.

We describe the minimization algorithm by fusion for acyclic automata, and for a more general class of automata, namely those recognizing regular languages having polynomially bounded number of words.

Finally, we consider briefly incremental algorithms for building minimal automata for finite sets of words. We consider the updating of a minimal automaton when a word is added or removed.

Index