DESCRIPTION AND ANALYSIS OF A BOTTOM-UP DFA MINIMIZATION ALGORITHM

JORGE ALMEIDA AND MARC ZEITOUN

ABSTRACT. We establish linear-time reductions between the minimization of a deterministic finite automaton (DFA) and the conjunction of 3 subproblems: the minimization of a strongly connected DFA, the isomorphism problem for a set of strongly connected minimized DFAs, and the minimization of a connected DFA consisting in two strongly connected components, both of which are minimized. We apply this procedure to minimize, in linear time, automata whose nontrivial strongly connected components are cycles.

1. INTRODUCTION

Finite automata have been successfully used in numerous fields of computer science such as pattern matching, compilation, natural language processing, databases, system verification. They can represent a broad range of objects, from dictionaries to models of transition systems. Properties expressed in high-level description formalisms must also often be "compiled" into automata before algorithms can be applied. For real-world applications, such automata may have a huge number of states, and reducing their size often proves to be crucial for subsequent treatment. Finite automata on finite words have a minimal, canonical representation with respect to the language they determine. This paper focuses on the process to compute this minimal representation, called *minimization*.

Under the usual assumption that letters and states are represented by integers that can be compared in $O(1)$-time, the best-known algorithm for minimizing a deterministic finite automaton (DFA) is Hopcroft's [11], with a $O(\ell m \log m)$ worst case time complexity where ℓ is the number of letters and m the number of states (see [10, 13, 3] for complexity analyses). Brzozowski's algorithm [5] works theoretically in exponential time, but has in practice a surprisingly good behavior (see [7]).

Minimization algorithms usually start from the equivalence separating final and non-final states, and refine it until stabilization occurs. In this paper, starting from the equality relation, we merge states which are detected to be equivalent. We reduce the minimization problem of a DFA to subproblems involving its strongly connected components on one hand, and its directed acyclic structure on the other hand. More precisely, for any function f such that $f(n)/n$ is nondecreasing, we show that the minimization problem can be solved in time $O(f(d + \ell))$, where ℓ still denotes the number of input symbols and d is the number of transitions (which can be smaller than ℓm since the algorithm can deal with incomplete automata), if and only if three subproblems have the same worst case complexity. These subproblems are (1) the minimization of a strongly connected DFA (2) the computation of isomorphisms between strongly connected, minimized DFAs, and (3) the minimization of connected DFAs having exactly two strongly connected components, both of which are already minimized. The reduction is presented by the generic algorithm 1 using subroutines solving the subproblems, which will be explained in detail later.

Using pattern matching techniques, we obtain as an application a $O(d+\ell)$-time minimization algorithm for automata whose nontrivial strongly connected components are cycles (this particular application was announced, without proof, in [2]). This extends Revuz's minimization algorithm [14] for acyclic DFAs, which was designed to compress dictionaries and works in $O(d + \ell)$-time with $O(m + \ell)$ memory. Other algorithms for minimizing acyclic DFAs in linear time [17] or to maintain a minimal DFA after adjunction of one word to its language have also been developed, see *e.g.*, [6].

Algorithm 1 Minimization algorithm (outline)

1: **procedure** MINIMIZE(Automaton $\mathcal{A}$)
2: $\quad$ $X \leftarrow$ ZEROHEIGHT($\mathcal{A}$)
3: $\quad$ **while** $X \neq \emptyset$ **do**
4: $\quad\quad$ MINIMIZESCC(X)
5: $\quad\quad$ MERGEISOMORPHICSCC(X)
6: $\quad\quad$ WRAP($\mathcal{A}, X$)
7: $\quad\quad$ $X \leftarrow$ NEXTHEIGHT($\mathcal{A}, X$)
8: $\quad$ **end while**
9: **end procedure**

---

2. AUTOMATA AND DATA STRUCTURES

We work on a finite alphabet $A = \{0, \ldots, \ell-1\}$ with $\ell \geq 2$ letters. We denote by A^* the free monoid generated by A, and by $|x|$ the length of a word $x \in A^*$. We assume that A is known, and can be used as index set for arrays. In Section 4, we also use the usual total order on A, viewed as a set of integers. A *deterministic finite automaton (DFA)* is a tuple $\mathcal{A} = (A, S, F, \delta, s_0)$ where A is the alphabet, S is a finite set of states, $F \subseteq S$ is the set of final states, $\delta : S \times A \to S$ is a partial mapping called the transition function, and s_0 is the initial state. We let $m = |S|$ be the number of states, $d = |\delta|$ be the number of transitions, and $n = d + \ell$.

The state $\delta(s, a)$, when it exists, is also written $s \cdot a$. We represent the transition $(s, a, s \cdot a)$ by $s \xrightarrow{a} s \cdot a$. We use the word *edge* to mean a transition. An automaton defines a directed graph with vertex set S and edge set $\{(s, s \cdot a) \mid s \in S, a \in A\}$. A *strongly connected component (scc)* of $\mathcal{A}$ is a strongly connected component of this graph, and $\mathcal{A}$ is *strongly connected* if so is its associated graph. We retain the terminology of [8]: a *strongly connected component (scc)* of a graph is an equivalence class for the mutual reachability relation. In particular, a vertex u such that the only path from u to u is empty is an scc, which is said to be *trivial*.

Given a state $s \in S$, the language recognized by $\mathcal{A}$ from s is the set $\mathcal{L}_{\mathcal{A}}(s) \subseteq A^*$ of words labeling a path from s to some final state. Two states s, t are *(Nerode) equivalent* if $\mathcal{L}_{\mathcal{A}}(s) = \mathcal{L}_{\mathcal{A}}(t)$. We write $[s]$ for the class of s in this equivalence. The *minimization* procedure consists in computing this equivalence relation. Merging the states of each class into a single state produces the minimal automaton recognizing the same language from the initial state, see [12]. A DFA is *minimal*, or *minimized*, if no two distinct states are equivalent. We are interested in the complexity of minimization in terms of the parameters ℓ, m, d and n.

We assume that the DFA $\mathcal{A}$ is accessible and co-accessible, that is, all states are reachable from s_0 and can reach a final state. Other states are useless regarding the accepted language, and removing them can be done in $O(d + m)$-time. Further, the initial state is irrelevant for the computation of equivalent states (it just serves for determining the initial state of the minimized DFA). For this reason, we drop the initial state, keeping in mind that we started from an accessible automaton, so that we have $m \leq d + 1$ and $O(d + m) = O(d)$.

For a class $\mathcal{C}$ of DFAs, we study the following problem.

$\underline{\mathcal{C}\text{-MINIMIZATION}}$ Minimizing a DFA of the class $\mathcal{C}$.

Input:: A finite deterministic automaton from $\mathcal{C}$.
Output:: Its minimal automaton given by the equivalence relation on states.

The automaton can be given by a matrix of $(S \cup \{\_\})^{S \times A}$ whose (s, a) entry is $s \cdot a$ if it is defined, or $\_$ otherwise. Using lists yields a smaller representation: for each $s \in S$, we are given a list of the form $(a_1, s_1, \ldots, a_k, s_k)$, with $a_i \in A$ and $s_i \in S$, describing all outgoing transitions $s \xrightarrow{a_i} s_i$. (These lists can be computed from the matrix in $O(\ell m)$-time.) A list $(a_1, s_1, \ldots, a_k, s_k)$ of $(A.S)^*$ is *sorted* if $a_1 < a_2 < \cdots < a_k$. We write $\mathrm{Out}(s)$ for the sorted list of outgoing transitions of $s \in S$.

We first sort outgoing transitions. Recall that, given a finite set X, one can sort a sequence $u_1, \ldots, u_k \in X^*$ in time $O(|u_1 \cdots u_k| + |X|)$ using radix-sort [1]. Applying this result to transitions of a DFA yields the following simple statement.

Lemma 2.1. *Let $\mathcal{A}$ be an accessible DFA (with an initial state). Given, for each state of $\mathcal{A}$, a list of all its outgoing transitions, one can compute in $O(n)$-time all sorted lists of outgoing transitions of states of $\mathcal{A}$, where $n = d + \ell$.*

Proof. For each list $\mathrm{Out}(s) = (a_1, s_1, \ldots, a_k, s_k)$, first build the transition list $((s, a_1, s_1), \ldots, (s, a_k, s_k))$. Since $\mathcal{A}$ is accessible, we have $m \leq d + 1$, so this step takes $O(d + m) = O(d)$-time. One then uses radix sort on the list of *all* such transitions to order them lexicographically according to the first two components, which costs $O(d + m) + O(d + \ell) = O(n)$. In the sorted list obtained, the transitions (s, a, t) with the same state s are consecutive and sorted according to the second component. It remains to scan this list to break it into pieces corresponding to the same state s, to build each of the sorted lists. Altogether, this requires $O(d + \ell) = O(n)$-time. $\qquad\square$

The complexity $O(d)$ is the best possible for minimization algorithms, since one needs to visit all transitions. Note that $O(n) = O(d)$ if each letter of A labels at least one transition. Therefore, by Lemma 2.1, one can start with a sorted list representation for DFAs. We assume that lists are doubly linked: one can access each of the predecessors of a state, individually, in $O(1)$-time.

3. A reduction for the minimization problem

3.1. **Minimizing acyclic DFAs.**
Our algorithm is inspired by Revuz's [14] for acyclic automata, which we briefly recall. Recall that $[t]$ denotes the class of the state t in the Nerode equivalence. We associate with each state s such that $\mathrm{Out}(s) = (a_1, s_1, \ldots, a_k, s_k)$ the tuple $\tau(s) = (\varepsilon_s, a_1, [s_1], \ldots, a_k, [s_k])$, where $\varepsilon_s = 1$ if $s \in F$ and $\varepsilon = 0$ otherwise. The algorithm first computes the *height* of each state, which is the length of the longest path to a final state. Note that equivalent states must have the same height. At stage $h = 0, 1, \ldots$, up to the maximal height H, the algorithm merges states of height h. Since $[s] = [t]$ if and only if $\tau(s) = \tau(t)$, radix sorting the words $\tau(s)$ for states of height h yields a list with equal words at consecutive places, which allows identifying equivalent states of height h. A minor complication is that using H times radix sort produces a complexity of $O(d + H\ell)$. This motivates the following statement, appearing in [14, Theorem 2 and page 187].

Lemma 3.1. *Given a set X and $u_1, \ldots, u_k \in X^*$, one can compute in time $O(|u_1 \cdots u_k|)$ the equality classes on $(u_1, \ldots, u_k)$, using an already allocated 0-initialized X-indexed array which is reset after the computation.*

Proof. Let $K = |u_1 \cdots u_k|$. We do *not* want X, which may be huge compared to K if many letters are unused, to appear in the complexity bound. In one scan, one rewrites the sequence $u_1, \ldots, u_k$ using only *consecutive* positive integer letters, thus obtaining words $u_1', \ldots, u_k'$, as follows. We store the encoding of $x \in X$ in $T[x]$, where T is the 0-initialized array. We replace the occurrence of each scanned letter x by its encoding $T[x]$ if x has already been encoded ($T[x] \neq 0$). Otherwise, we first increment the number of distinct letters already encountered, we assign the result to $T[x]$, and we push x on a stack (which therefore contains the nonzero entries of T). This rewriting requires $O(K)$ operations. The size of the alphabet of consecutive integers is $O(K)$, so applying radix-sort to $u_1' \ldots, u_k'$ determines equality classes in time $O(K)$ (since $u_i' = u_j'$ if and only if $u_i = u_j$). Finally, using the stack, one switches back to 0 all nonzero entries of T in time $O(K)$. $\qquad\square$

Using the algorithm of Lemma 3.1 instead of radix sort directly to minimize acyclic automata yields the desired $O(n)$ time complexity: $O(\ell)$ time is needed to allocate the 0-indexed array, and $K = O(d + m) = O(d)$ time to determine equality classes.

3.2. **The bottom-up minimization algorithm.**

3.2.1. *Description.* The same scheme applied to arbitrary DFAs brings additional difficulties. First, the notion of height has to be modified, since there may be paths of arbitrary length to a final state. We define the *height* of a state by considering each strongly connected component (scc) as a single state. This requires that one first compute the directed acyclic graph (DAG) of scc's of the automaton, which can be done in $O(d)$-time with Tarjan's algorithm [16, 8]. We maintain this DAG along the algorithm. (Expressions such as *an scc is below another scc* refer to the partial order induced by this DAG.) In the rest of the paper, we identify each scc with its set of states. We use an array of size m storing, for each state s, the number of its scc. Conversely for each scc, we record its list of states. We also use the same data structures for all equivalence relations, to have access in $O(1)$-time to the equivalence class of a state computed so far. To define heights, we assign weight 0 to an edge belonging to an scc and weight 1 to all other edges. The *weight* of a path is the sum of the weights of all edges in the path. The *height* of a state is then the maximal weight of some path to a final state, which is well defined. By definition, all states of a given scc have the same height.

One could compute the height along one traversal, as for a DAG, but the problem is that two states at different heights may well be equivalent. For instance, consider the automaton $s_1 \xrightarrow{a} s_0 \xrightarrow{a} s_0$ with both s_1, s_0 final. The height of s_1 is 1 and the height of s_0 is 0. However, s_1 is equivalent to s_0. We say that s_1 can be *wrapped* onto the scc of s_0. Formally, starting from an automaton with exactly two scc's C_0 and C_1, where C_1 is connected to C_0 and both C_0, C_1 are already minimized as individual automata (taking into account, for C_1, the transitions leading to C_0), wrapping consists in determining pairs of equivalent states of $C_1 \times C_0$. If such a pair exists, then C_0 is nontrivial, and every state of C_1 is equivalent to some state of C_0. In the example, our algorithm shall wrap s_1 onto the cycle s_0 (identifying s_0 and s_1). However, doing so changes the height of s_1 (from 1 to 0) and more generally, wrapping states may decrease the height of states that lie above them. The other difficulty is that both computations are linked: the height is needed to determine which state to wrap at some point in the algorithm, and one also needs to modify the heights after a wrapping.

To avoid recomputing the heights several times, we do not compute them beforehand, and we maintain information to determine on the fly, before stage k, which states must be treated at this stage. Nonetheless, to help understanding the computation on-the-fly, we give a description of how we would precompute the height of all states in $O(n)$-time: one assigns a *mass* to each state (stored in its data structure). A state with i outgoing 1-weighted edges has initially mass i. The mass of an scc is the maximal mass of its states. The mass of a state decreases during the execution of the algorithm. We record for each scc its number of states and its number of states of mass 0. Initially, these two numbers are equal only for minimal scc (in the DAG of scc). Each time we decrease the mass of a state, we check whether it reaches mass 0. If so, we increment the number of states having mass 0 for its scc. If this number reaches the total number of states in the scc, then the scc itself reaches mass 0, and we add it to a list of scc of mass 0.

Initially, we assign height 0 to all states in scc of mass 0. Further heights will be computed later on, at different steps of the algorithm. When states of height less than $h - 1$ have been treated, we need to compute the set of states of height h. These states are obtained, at that stage, as those belonging to scc's of mass 0. Then, these states are not considered anymore for the height computation (we remove the corresponding scc from the list of scc's of mass 0). Moreover, for each transition $t \xrightarrow{a} s$ ending in a state s to which we just assigned height h, we decrease the mass of state t by 1, increase the count of mass 0 states of its scc if t reaches mass 0, and put its scc in the list of scc's of mass 0 if this count reaches its number of states. Observe that each transition is considered at most once, so that the overall time complexity, for the height computation, is $O(d)$.

An outline of a generic minimization algorithm is described in Algorithm 1. It uses three subroutines, MINIMIZESCC, MERGEISOMORPHICSCC and WRAP, assumed to be given, and described below. It computes a sequence of automata $\mathcal{A}_{-1} = \mathcal{A}, \mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_H$ such that $\mathcal{A}_H$ is the minimal automaton of $\mathcal{A}$. Stage $h \in [0, H]$ merges equivalent states of height h of $\mathcal{A}_{h-1}$ to produce $\mathcal{A}_h$. The automaton $\mathcal{A}_h$ is obtained at the end of the hth iteration of the main loop of Algorithm 1, after that all merging of states of height at most h will have been performed.

The variable X always holds (states of) a subset of the set of scc's of the current automaton, for which merging should occur at lines 4–6. It is initialized, at line 2, with all scc's of mass 0 (precomputed by Tarjan's algorithm). At line 7, it receives the candidate states for merging at the next iteration, that is, the part of $\mathcal{A}_{h-1}$ consisting of states of height h, at that stage.

Let us explain the calls of lines 4–7. The first two of them only merge states of X. The call of line 6 possibly merges states of the part of $\mathcal{A}_{h-1}$ not yet treated (cf. Fig. 1) with states of X.

The call MINIMIZESCC(X) minimizes separately each scc $C_1, \ldots, C_p$ of X, taking also into account the transitions going to an scc below X. Let C be an scc of X. Some states of C may have transitions to scc's below C in the DAG of scc. However, since the algorithm proceeds bottom-up, the part of the automaton below C is already minimized when C is considered. Therefore, one can first use a minimization algorithm on C as if it were an automaton by itself, not considering the transitions falling below C. This gives us a partition into equivalence classes $\sim_1$. We then refine this partition according to the equivalence $\sim_2$ induced by the transitions going below C: two states are $\sim_2$-equivalent if and only if they reach the same states of the part already treated by the algorithm, by the same transition labels. The call MINIMIZESCC(X) computes the equivalence $\sim_1 \cap \sim_2$. To refine $\sim_1$, once computed, by $\sim_2$ we associate to a state $s \in C$ with transitions $s \xrightarrow{a_i} s_i$ $(1 \leq i \leq k)$ falling below C the word $(\varepsilon_s, [s]_1, a_1, [s_1]_2, \ldots, a_k, [s_k]_2)$. By Lemma 3.1, one can sort these words in time $O(d_i)$ where d_i is the number of such transitions, assuming that an array of size $\max(2, m, \ell)$ has been allocated at the beginning of the algorithm, once for all. The overall cost is therefore $O(d + \max(2, m, \ell)) = O(n)$.
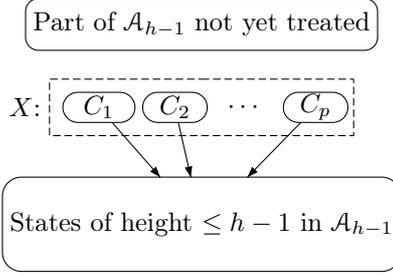
FIGURE 1. Automaton $\mathcal{A}_{h-1}$ during the algorithm

Let $C'_1, \ldots, C'_p$ be the scc's of X after line 4. The call MERGEISOMORPHICSCC(X) merges all scc's in X that are isomorphic, and X gets modified accordingly: it then contains a set of representatives $\{C'_{i_1}, \ldots, C'_{i_j}\}$, $j \leq p$, of isomorphic scc's (so that we make coarser the equivalence on states computed so far). As in the previous case, we then have to refine this partition according to the transitions falling below X.

The call WRAP$(\mathcal{A}, X)$ occurs when X is already minimized. It consists in possibly identifying states of scc's that are located *above* one of the C'_{i_k} to an already minimized scc of X, as explained earlier in the example $s_1 \xrightarrow{a} s_0 \xrightarrow{a} s_0$. Note that if some state t above X in the DAG of scc's is equivalent to some state in C'_{i_k}, then all states belonging to scc's between t and X are also equivalent to some state in C'_{i_k}. The procedure WRAP$(\mathcal{A}, X)$ precisely merges these scc to X. Again, to validate that two states are equivalent, we have to take into account transitions falling below X.

The last step in each iteration of the main loop, line 7, is the update $X \leftarrow$ NEXTHEIGHT$(\mathcal{A}, X)$ computing the set of scc's to consider during the next iteration. The height of states, as defined above, is not invariant through the call WRAP$(\mathcal{A}, X)$: a state may have its height lowered. For that reason, we update in the call NEXTHEIGHT$(\mathcal{A}, X)$ the weights of edges as follows: all edges leading to a state of X are assigned weight 0 (instead of 1 previously). The weights of all other edges remain unchanged. We then compute, only at this point, the mass of each state having an outgoing edge whose weight has been affected. The states reaching mass 0 are put in a list. They are exactly those we need for the next iteration, and are returned by the call to NEXTHEIGHT. Since the weight of each edge is modified at most once, the overall cost of all calls to NEXTHEIGHT is $O(d)$.

3.2.2. *Correctness.* We prove that Algorithm 1 indeed computes the minimal automaton. First, we only identify equivalent states, since merging is only done by the minimization subroutines of lines 4–6 assumed to be correct.

We have to prove that, whenever two states are equivalent, they are merged in the last automaton $\mathcal{A}_H$. Arguing by contradiction, assume that two equivalent states s, t of $\mathcal{A}$ have not been merged and suppose that the pair (s, t) is minimal in the DAG of scc's for this property. That is, if s' is below s, t' is below t, and s', t' are equivalent and distinct, then $s = s'$ and $t = t'$. States s and t cannot occur in the same value of the variable X since, otherwise, they would be merged at line 4 if they belong to the same scc, or at line 5 otherwise. Suppose that s is the first to occur in the value of X. Then, two cases may arise. One case is that t is wrapped to another state at line 6 while s belongs to X. This is impossible since, as X has been previously minimized (lines 4 and 5), no state of $X \setminus \{s\}$ is equivalent to s. It remains the case where s, t occur in X in two different iterations i_s, i_t of the main loop, with $i_s < i_t$. Observe that, for every letter a such $s \cdot a$ falls below s or $t \cdot a$ falls below t, since $s \cdot a$ and $t \cdot a$ are equivalent, by the minimality of the pair (s, t), they will be merged by the algorithm before s appears in the value of X. Since the remaining edges do not intervene in determining when t will appear in X, it follows that s and t will be found in the same value of X, a case which has already been excluded. This proves that Algorithm 1 is correct.

3.2.3. *Reductions.* We have isolated in our algorithm three subroutines to merge equivalent states in three different situations: (1) *minimizing a strongly connected component*, (2) *merging isomorphic strongly connected components*, and (3) *wrapping*. We formulate these subproblems for a class $\mathcal{C}$ of strongly connected DFAs.

$\mathcal{C}$-MSCA Minimizing strongly connected automata of $\mathcal{C}$.

Input:: A strongly connected DFA belonging to $\mathcal{C}$.

Output:: Its minimal automaton given by the equivalence relation on states.

C-MMSCC Merging minimized DFAs from C which are strongly connected.
Input:: A set of minimized and strongly connected DFAs $(\mathcal{A}_i)_{1 \leq i \leq m}$ of C.
Output:: (a) A partition $\bigcup_{j \in J} I_j$ of $[1, m]$ such that $\mathcal{A}_p, \mathcal{A}_q$ are isomorphic if and only if p, q are in the same I_j. (b) A representative of each class. (c) For each element in a class different from the chosen representative, an isomorphism to the representative.

C-WRAPPING Wrapping on a minimized scc of C.
Input:: A DFA consisting of a minimized scc $\mathcal{A}_0$ from C of height 0 and an scc $\mathcal{A}_1$ from C of height 1.
Output:: Its minimal DFA given by the equivalence relation on states.

For a class C of DFAs, let $\mathsf{dfa}(\mathcal{C})$ be the class of DFAs whose scc's are in C for some choice of final states.

If there is an $O(f(n))$-time algorithm for $\mathsf{dfa}(\mathcal{C})$-MINIMIZATION, then C-MSCA, C-MMSCC and C-WRAPPING also have an $O(f(n))$-time solution. This is clear for C-MSCA and C-WRAPPING which are the minimization problem on particular instances. For C-MMSCC, assume we are given several strongly connected nontrivial automata $(\mathcal{A}_i)_{1 \leq i \leq m}$ from C. Let $a, b \notin A$ be two distinct letters. Choose a state t_i in each $\mathcal{A}_i$ and consider the DFA $\mathcal{A}$ built by adding states $s_1, \ldots, s_m$ to the disjoint union of the automata $\mathcal{A}_i$, where the s_i's are new states, and transitions $s_i \xrightarrow{a} s_{i+1}$ and $s_i \xrightarrow{b} t_i$ (see Fig. 2). The disjoint union of automata $\mathcal{A}_i = (A_i, S_i, F_i, \delta_i)$, $1 \leq i \leq n$, is the automaton $\mathcal{A} = (\bigcup A_i, \biguplus S_i, \biguplus F_i, \biguplus \delta_i)$ (whose state set is the disjoint union $\biguplus S_i$). Obviously, $\mathcal{A}$ is in $\mathsf{dfa}(\mathcal{C})$ and has $O(D)$ transitions, where D
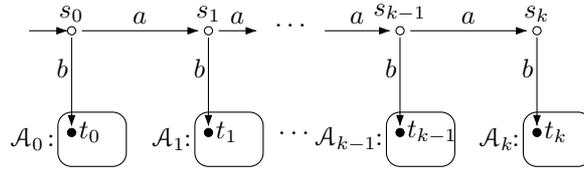

FIGURE 2. Merging minimal sc-automata

is the total number of transitions of all $\mathcal{A}_i$'s. Minimizing $\mathcal{A}$ exactly merges those $\mathcal{A}_i$ that are isomorphic, since the $\mathcal{A}_i$'s are minimal.

3.2.4. *Complexity.* We have shown in Section 3.2.3 that if $\mathsf{dfa}(\mathcal{C})$-MINIMIZATION can be solved in $O(f(n))$ time, then so can C-MMSCC, C-MSCA, and C-WRAPPING.

Conversely, we use the algorithm of Section 3.2.1, which calls subroutines solving these subproblems in order to solve $\mathsf{dfa}(\mathcal{C})$-MINIMIZATION. Assume that the subroutines run in time $O(f(n))$, where $f(x+y) \geq f(x) + f(y)$ (which is the case, *e.g.*, if $f(n)/n$ is nondecreasing). The time complexity for minimizing the automaton is the sum of (1) the complexity of all calls to the three subroutines, (2) the overhead to compute heights, and (3) the overhead to refine relations (*e.g.*, to compute $\sim_1 \cap \sim_2$). For (1), each subroutine is called several times, on subautomata of sizes $n_1, \ldots, n_p$, where $\sum n_i = n$, yielding an overall complexity of $f(n_1) + \cdots + f(n_p) = O(f(n))$ by the assumption on f. We have seen in Section 3.2.1 that the complexity for (2) is $O(n) = O(f(n))$. Finally, for (3), note that the refinements occur at most three times on each state (after the calls of lines 6, 4 and 5). The equivalence is computed by storing the class `Class[s]` of state s and the list `States[c]` of states of class c, using arrays. Merging two states s and t amounts to removing, say s, from `States[Class[s]]`, appending it to `States[Class[t]]`, and changing the value of `Class[s]`. These operations can be done in $O(1)$-time (Implementing the removal in $O(1)$-time is done by maintaining a pointer for each state s to its position in the list `States[Class[s]]`.) Hence the overall complexity is $O(f(n))$. We can state our main result.

Theorem 3.2. *Let* C *be a class of strongly connected DFAs containing the trivial DFAs (one state, no edge) and let* f *be a function such that* $f(n)/n$ *is nondecreasing. Then, the* $\mathsf{dfa}(\mathcal{C})$-MINIMIZATION *problem is solvable in* $O(f(n))$-*time if and only if* C-MMSCC, C-MSCA, *and* C-WRAPPING *are solvable in* $O(f(n))$-*time.*

We have to include trivial components in C for the wrapping, since we need to be able to wrap a single state to a (nontrivial) scc. Note that if we take for C the class of trivial scc's, we reobtain the linear complexity for the minimization of acyclic automata [14]. In the next section we apply Theorem 3.2 to a larger subclass of automata which one can still minimize in time $O(n)$.

4. Minimizing disjoint-cycle automata

A *disjoint-cycle automaton* is an automaton such that all strongly connected components are (possibly trivial) cycles. In other words, two cycles on distinct sets of vertices share no vertices. One can detect whether an automaton is disjoint-cycle, by checking for each state that at most one outgoing edge remains in the same strongly connected component.

We show in this section that the MMSCC, MSCA and WRAPPING problems for strongly connected components of this class are solvable in $O(n)$-time (Lemmas 4.2, 4.3 and 4.4 below). In view of Theorem 3.2, this will entail the following result, announced in [2].

Theorem 4.1. *One can minimize a disjoint-cycle automaton on ℓ letters with d transitions in time $O(d + \ell)$.*

The fact that scc's of a disjoint-cycle automaton are cycles allows us to work on words instead of working directly on automata. Recall that the conjugates of a word $b_1 \cdots b_p$ (where b_i are letters) are the words of the form $b_i b_{i+1} \cdots b_p \cdot b_1 \cdots b_{i-1}$. A circular word is a conjugation class. Slightly abusing notation, we represent a circular word by any word of its class.

We can associate to the cycle $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_k} s_0$ the circular word $(\varepsilon_0, a_0)(\varepsilon_1, a_1) \cdots (\varepsilon_k, a_k)$ where $\varepsilon_i = 1$ if s_i is final and $\varepsilon_i = 0$ otherwise. Conversely, from such a circular word, one can recover a unique cycle (up to the name of the states).

Lemma 4.2. MSCA *is solvable in linear time for disjoint-cycle automata.*

Proof. Recall that the primitive root of a word w is the shortest word r such that $w = r^k$ for some k. It is easy to see that minimizing a cycle $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} s_0$ amounts to finding a primitive root of its associated circular word: this primitive root is itself a circular word, and the cycle associated to it is the minimal automaton of the original cycle. It is classical that this computation can be performed in linear time (see *e.g.* [9] for instance). $\square$

Lemma 4.3. MMSCC *is solvable in linear time for disjoint-cycle automata.*

Proof. The problem can be formulated as follows in terms of circular words: we are given k circular words and we want to merge them into equality classes in linear time, with respect to the sum of their lengths. For that purpose, we compute for each circular word its associated Lyndon word, that is its smallest representant, in the lexicographic order. (This is the place where we use the fact that the alphabet is ordered.) Since we assumed that comparisons take linear time, the computation of the associated Lyndon word can be performed in linear time for each word, in terms of its length [4, 15]. It remains to group in classes circular words having the same Lyndon word, which can be done using Lemma 3.1. $\square$

Lemma 4.4. WRAPPING *is solvable in linear time for disjoint-cycle automata.*

Proof. Let $\mathcal{A}$ be an automaton having a single minimal scc in the DAG of strongly connected components. We distinguish two cases, depending on whether the highest scc is trivial (Fig. 3 (a)) or not (Fig. 3 (b)). In case (a), s can be wrapped on the cycle if and only if s is equivalent to t, that is if $a = b$. In case (b),
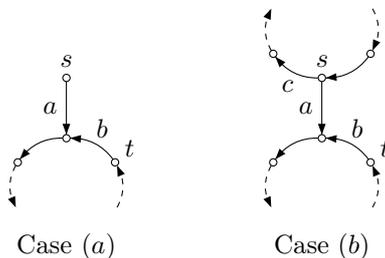

Case (a) Case (b)

FIGURE 3. Two cases for the WRAPPING problem

the only possible wrapping would identify s and t, hence $a = b$. Therefore, there should exist a transition from t labeled c, where c labels the transition from s inside its scc. This is not the case since, as the automaton is deterministic, we have $c \neq a$, and the only transition from t is labeled by a. Hence no wrapping occurs in this case. $\square$

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1975. Second printing.

[2] J. Almeida and M. Zeitoun. The equational theory of ω-terms for finite $\mathcal{R}$-trivial semigroups. In *Proceedings of Semigroups and Languages (Lisbon 2002)*, pages 1–23. World Scientific, 2004.

[3] D. Beauquier, J. Berstel, and Ph. Chrétienne. *Éléments d'Algorithmique*. Masson, 1992. In French. http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.html.

[4] K. S. Booth. Lexicographically least circular substrings. *Inform. Process. Lett.*, 10:240–242, 1980.

[5] J. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. *MRI Symposia Series*, 12:529–561, 1962. Polytechnic Press, Polytechnic Institute of Brooklyn.

[6] R. Carrasco and M. Forcada. Incremental construction and maintenance of minimal finite-state automata. *Computational Linguistics*, 28(1):207–216, 2002.

[7] J.-M. Champarnaud and D. Ziadi. Canonical derivatives, partial derivatives and finite automaton constructions. *Theor. Comput. Sci.*, 289(1):137–163, 2002.

[8] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, 2001.

[9] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994. With a preface by Zvi Galil.

[10] D. Gries. Describing an algorithm by Hopcroft. *Acta Inform.*, 2:97–109, 1973.

[11] J. E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi, editor, *Theory of machines and computations (Proc. Internat. Sympos., Technion, Haifa, 1971)*, pages 189–196. Academic Press, 1971.

[12] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, 2000.

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

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

[15] Y. Shiloach. Fast canonization of circular strings. *J. Algorithms*, 2:107–121, 1981.

[16] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

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

Centro de Matemática e Departamento de Matemática Pura, Faculdade de Ciências,, Universidade do Porto, Rua do Campo Alegre, 687, 4169-007 Porto, Portugal.

E-mail address: jalmeida@fc.up.pt

LaBRI, Université Bordeaux & CNRS UMR 5800. 351 cours de la Libération, 33405 Talence Cedex, France.

E-mail address: mz@labri.fr