Theoretical Computer Science 411 (2010) 3404–3413

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

An $n \log n$ algorithm for hyper-minimizing a (minimized) deterministic automaton☆

Markus Holzer[a], Andreas Maletti[b,*]

[a] Institut für Informatik, Universität Giessen, Arndtstr. 2, 35392 Giessen, Germany

[b] Departament de Filologies Romàniques, Universitat Rovira i Virgili, Av. Catalunya 35, 43002 Tarragona, Spain

ARTICLE INFO

Keywords:
Deterministic finite automaton
Minimization
Lossy compression

ABSTRACT

We improve a recent result [A. Badr, Hyper-minimization in $O(n^2)$, Internat. J. Found. Comput. Sci. 20 (4) (2009) 735–746] for hyper-minimized finite automata. Namely, we present an $O(n \log n)$ algorithm that computes for a given deterministic finite automaton (dfa) an almost-equivalent dfa that is as small as possible—such an automaton is called hyper-minimal. Here two finite automata are almost-equivalent if and only if the symmetric difference of their languages is finite. In other words, two almost-equivalent automata disagree on acceptance on finitely many inputs. In this way, we solve an open problem stated in [A. Badr, V. Geffert, I. Shipman, Hyper-minimizing minimized deterministic finite state automata, RAIRO Theor. Inf. Appl. 43 (1) (2009) 69–94] and by Badr. Moreover, we show that minimization linearly reduces to hyper-minimization, which shows that the time-bound $O(n \log n)$ is optimal for hyper-minimization. Independently, similar results were obtained in [P. Gawrychowski, A. Jeż, Hyper-minimisation made efficient, in: Proc. 34th Int. Symp. Mathematical Foundations of Computer Science, in: LNCS, vol. 5734, Springer, 2009, pp. 356–368].

1. Introduction

Early studies in automata theory revealed that nondeterministic and deterministic finite automata are equivalent [1]. However, nondeterministic automata can be exponentially more succinct [2,3] (with respect to the number of states). In fact, finite automata are probably best known for being equivalent to right-linear context-free grammars, and thus, for capturing the lowest level of the Chomsky-hierarchy, which is the family of regular languages. Over the last 50 years, a vast literature documenting the importance of finite automata as an enormously valuable concept has been developed. Although, there are a lot of similarities between nondeterministic and deterministic finite automata, one important difference is that of the minimization problem. The study of this problem also dates back to the early beginnings of automata theory. It is of practical relevance because regular languages are used in many applications, and one may like to represent the languages succinctly. While for nondeterministic automata the computation of an equivalent minimal automaton is PSPACE-complete [4] and thus highly intractable, the corresponding problem for deterministic automata is known to be effectively solvable in polynomial time [5]. An automaton is minimal if every other automaton with fewer states disagrees on acceptance for at *least one* input.

Minimizing deterministic finite automata (dfa) is based on computing an equivalence relation on the states of the automaton and collapsing states that are equivalent. Here two states $p, q \in Q$, where Q is the set of states of the automaton

☆ This is an extended and revised version of [M. Holzer, A. Maletti, An $n \log n$ algorithm for hyper-minimizing states in a (minimized) deterministic automaton, in: Proc. 14th Int. Conf. Implementation and Application of Automata, in: LNCS, vol. 5642, Springer, 2009, pp. 4–13].

* Corresponding author. Tel.: +34 977 558 382; fax: +34 977 558 386.
E-mail addresses: holzer@informatik.uni-giessen.de (M. Holzer), andreas.maletti@urv.cat (A. Maletti).

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.05.029

under consideration, are equivalent, if the automaton starting its computation in state p accepts the same language as the automaton if q is taken as the start state. Minimization of two equivalent dfa leads to minimal dfa that are isomorphic up to the renaming of states. Hence, minimal dfa are unique. This yields a nice characterization: A dfa M is *minimal* if and only if in M: (i) there are no unreachable states and (ii) there is no pair of different but equivalent states.

The computation of this equivalence can be implemented in a straightforward fashion by repeatedly refining partitions starting with the partition that groups accepting and rejecting states together [5]. This yields a polynomial-time algorithm of $O(n^2)$. Hopcroft's algorithm [6] for minimization slightly improves the naïve implementation to a running time of $O(m \log n)$ with $m = |Q \times \Sigma|$ and $n = |Q|$, where Σ is the alphabet of input symbols of the finite automaton. It is up to now the best known minimization algorithm for dfa in general. Recent developments have shown that this bound is tight for Hopcroft's algorithm [7,8]. Thus, minimization can be seen as a form of lossless compression that can be done effectively while preserving the accepted language exactly.

Recently, a new form of minimization, namely hyper-minimization, was studied in the literature [9–11]. There the minimization or compression is done while giving up the preservation of the semantics of finite automata; i.e., the accepted language. It is clear that the semantics cannot vary arbitrarily. A related minimization method based on cover automata is presented in [12,13]. Hyper-minimization [9–11] allows the accepted language to differ in acceptance on a *finite number* of inputs, which is called *almost-equivalence*. Thus, hyper-minimization aims to find an almost-equivalent dfa that is as small as possible. Here an automaton is *hyper-minimal* if every other automaton with fewer states disagrees on acceptance for an *infinite* number of inputs.

In [9] basic properties of hyper-minimization and hyper-minimal dfa are investigated. Most importantly, a characterization of hyper-minimal dfa is given, which is similar to the characterization of minimal dfa mentioned above. Namely, a dfa M is *hyper-minimal* if and only if in M: (i) there are no unreachable states, (ii) there is no pair of different but equivalent states, and (iii) there is no pair of different but almost-equivalent states such that at least one of them is a preamble state. Here a state is called a *preamble state* if it is reachable from the start state by a *finite* number of inputs, only. Otherwise the state is called a *kernel state*. These properties allow a structural characterization of hyper-minimal dfa. Roughly speaking, the kernels (all states that are kernel states) of two almost-equivalent hyper-minimal automata are isomorphic in the standard sense, and their preambles are also isomorphic except for acceptance values. Thus, it turns out that hyper-minimal dfa are not necessarily unique. Nevertheless, it was shown in [9] that hyper-minimization can be done in time $O(mn^2)$, where $m = |\Sigma \times Q|$ and $n = |Q|$. For a constant alphabet size this gives an $O(n^3)$ algorithm. Later, the bound was improved [10,11] to $O(mn)$. In this paper we improve this upper bound further to $O(m \log n)$. If the alphabet size is constant, then this yields an $O(n \log n)$ algorithm. In addition, we argue that this is reasonably good because any upper bound $t(n) = \Omega(n)$ for hyper-minimization implies that (classical) minimization can be done within $t(n)$. To this end, we linearly reduce minimization to hyper-minimization. Similar results were independently obtained in [14].

The results of this paper were first reported in [15]. This version contains the full, detailed proofs of the claims, a more elaborate example, and a few minor corrections. The paper is organized as follows: In the next section we introduce the necessary notation. Then in Section 3 we first describe the general background needed to perform hyper-minimization, namely identifying kernel states, computing almost-equivalent states, and finally merging almost-equivalent states. Next we present a running example, and show how to implement these three sub-tasks in time $O(m \log n)$. The formal time-complexity and correctness proofs are presented in Sections 4 and 5. In Section 4 we also show the linear reduction from minimization to hyper-minimization. Finally we summarize our results and state some open problems.

2. Preliminaries

The set of nonnegative integers is denoted by $\mathbb{N}$. For sets $S \subseteq A$, $T \subseteq B$, and a function $h\colon A \to B$, we write $h(S) = \{h(s) \mid s \in S\}$ and $h^{-1}(T) = \{s \in A \mid h(s) \in T\}$. A relation on S is a subset of $S \times S$. The relation R on S is more refined than the relation R' on S if $R \subseteq R'$. A relation on S is an equivalence relation if it is reflexive, symmetric, and transitive. In the usual way, each equivalence relation induces a partition of S, which is a set of disjoint subsets of S such that their union is S. Conversely, every partition of S induces an equivalence relation on S.

Let S and T be sets. Their symmetric difference $S \ominus T$ is $(S \setminus T) \cup (T \setminus S)$. The sets S and T are almost-equal if $S \ominus T$ is finite. An alphabet Σ is a finite set. The cardinality of an alphabet Σ is denoted by $|\Sigma|$. The set of all strings over Σ is Σ^*, of which the empty string is denoted by ε. The concatenation of strings $u, w \in \Sigma^*$ is denoted by the juxtaposition uw, and $|w|$ denotes the length of a word $w \in \Sigma^*$. A deterministic finite automaton (dfa) is a tuple $M = (Q, \Sigma, q_0, \delta, F)$ where Q is the finite set of states, Σ is the alphabet of input symbols, $q_0 \in Q$ is the initial state, $\delta\colon Q \times \Sigma \to Q$ is the transition function, and $F \subseteq Q$ is the set of final states. The transition function δ extends to $\delta\colon Q \times \Sigma^* \to Q$ as follows: $\delta(q, \varepsilon) = q$ and $\delta(q, \sigma w) = \delta(\delta(q, \sigma), w)$ for every $q \in Q$, $\sigma \in \Sigma$, and $w \in \Sigma^*$. The dfa M recognizes the language $L(M) = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}$.

Let $\simeq$ be an equivalence relation on Q. It is a congruence if $\delta(p, \sigma) \simeq \delta(q, \sigma)$ for every $p \simeq q$ and $\sigma \in \Sigma$. Two states $p, q \in Q$ are equivalent, denoted by $p \equiv q$, if $\delta(p, w) \in F$ if and only if $\delta(q, w) \in F$ for every $w \in \Sigma^*$. Note that $\equiv$ is the coarsest (i.e., least refined) congruence that respects F (i.e., a final state cannot be congruent to a nonfinal state). The dfa M is minimal if it does not have equivalent states. The notion "minimal" is justified by the fact that no dfa with fewer states also recognizes $L(M)$ if M is minimal. A minimal dfa that is equivalent to M can efficiently be computed using Hopcroft's algorithm [6], which runs in time $O(m \log n)$ where $m = |Q \times \Sigma|$ and $n = |Q|$.

Algorithm 1 Overall structure of the hyper-minimization algorithm of [10,11].

Require: a dfa M
Return: a hyper-minimal, almost-equivalent dfa

```
   M ← Minimize(M)                          // Hopcroft's algorithm; O(m log n)
2: K ← ComputeKernel(M)                     // compute the kernel states; see Section 3.1
   ∼ ← AEquivalentStates(M)                 // compute almost-equivalence; see Section 3.2
4: M ← MergeStates(M, K, ∼)                 // merge almost-equivalent states; O(m)
   return M
```

In the following, let M be minimal. We recall a few central notions from [9] next. A state $q \in Q$ is a kernel state if $q = \delta(q_0, w)$ for infinitely many $w \in \Sigma^*$. Otherwise q is a preamble state. We denote the set of kernel and preamble states by $\mathrm{Ker}(M)$ and $\mathrm{Pre}(M)$, respectively. For two states $p, q \in Q$ we write $p \to q$ if there exists a $w \in \Sigma^*$ such that $\delta(p, w) = q$. They are strongly connected, denoted by $p \leftrightarrow q$, if $p \to q$ and $q \to p$. Note that $p \leftrightarrow p$ and $q_0 \to q$ for every $q \in Q$ because M is minimal. Two different, strongly connected states $p \leftrightarrow q$ are also kernel states because $q_0 \to p \to q \to p$. Finally, $q \in Q$ is a center state if $\delta(q, w) = q$ for some nonempty string $w \in \Sigma^*$. In other words, a center state is nontrivially strongly connected to itself.

3. Hyper-minimization

Minimization, which yields an equivalent dfa that is as small as possible, can be considered as a form of lossless compression. Sometimes the compression rate is more important than the preservation of the semantics. This leads to the area of lossy compression where the goal is to compress even further at the expense of errors (typically with respect to some error profile). Our error profile is very simple here: we allow a finite number of errors. Consequently, we call two dfa M_1 and M_2 *almost-equivalent* if their languages $L(M_1)$ and $L(M_2)$ are almost-equal. A dfa that admits no smaller almost-equivalent dfa is called *hyper-minimal*. Hyper-minimization [9–11] aims to find an almost-equivalent hyper-minimal dfa. In [14] hyper-minimization is also discussed for a more refined error profile, in which the length of the error-words can be restricted.

Recall that $M = (Q, \Sigma, q_0, \delta, F)$ is a minimal dfa. In addition, let $m = |Q \times \Sigma|$ and $n = |Q|$. The contributions [9–11,14] report hyper-minimization algorithms for M that run in time $O(mn^2)$, $O(mn)$, and $O(m \log n)$, respectively. Note that [14] was obtained independently from our research reported here. Our aim was to develop a hyper-minimization algorithm that runs in time $O(m \log n)$.

Let us start with the formal development. Roughly speaking, minimization aims to identify equivalent states, and hyper-minimization aims to identify almost-equivalent states, which we define next.

Definition 1 (*cf. [9, Definition 2.2]*)**.** For all states $p, q \in Q$, we say that p and q are *almost-equivalent*, denoted by $p \sim q$, if there exists $k \geq 0$ such that $\delta(p, w) = \delta(q, w)$ for every $w \in \Sigma^*$ with $|w| \geq k$.

The overall structure of the hyper-minimization algorithm of [10,11] is presented in Algorithm 1. Note that compared to [10,11], we exchanged lines 2 and 3. Minimize refers to the classical minimization, which can be implemented to run in time $O(m \log n)$ using Hopcroft's algorithm [6]. The procedure MergeStates is described in [9–11], where it is also proved that it runs in time $O(m)$. We present their algorithm (see Algorithm 2) and the corresponding results next. Roughly speaking, merging a state p into another state q denotes the usual procedure of redirecting in M all incoming transitions of p to q. In addition, if p was the initial state, then q is the new initial state. Formally, for every $\overline{\delta}\colon Q \times \Sigma \to Q$ and $p_0, p, q \in Q$ we define $\mathrm{merge}(\overline{\delta}, p_0, p, q) = (\delta', p_0')$ where for every $q' \in Q$ and $\sigma \in \Sigma$

$$\delta'(q', \sigma) = \begin{cases} q & \text{if } \overline{\delta}(q', \sigma) = p \\ \overline{\delta}(q', \sigma) & \text{otherwise} \end{cases} \quad \text{and} \quad p_0' = \begin{cases} q & \text{if } p_0 = p \\ p_0 & \text{otherwise.} \end{cases}$$

Clearly, the state p could now be deleted because it cannot be reached from the initial state anymore; i.e., $\delta'(p_0', w) \neq p$ for all $w \in \Sigma^*$. Observe, that although state p could have been a final state, the status of state q with respect to finality (membership in F) is *not* changed.

Whenever we discuss algorithms, we generally assume that the preconditions (Require) are met. If we call a procedure in one of our algorithms, then we will argue why the preconditions of that procedure are met.

Theorem 2 (*[9, Section 4]*)**.** *Algorithm 2 returns a hyper-minimal dfa that is almost-equivalent to M in time $O(m)$.*

Proof (*Sketch*)**.** The correctness is proved in detail in [9]. Globally, the selection process runs in time $O(n)$ if the almost-equivalence is supplied as a partition. Then an iteration over the transitions can perform the required merges in time $O(m)$. Since the surviving state of a merge is never merged into another state, each transition is redirected at most once. In fact, if the merge is implemented by a pointer redirect, then Algorithm 2 can be implemented to run in time $O(n)$. □

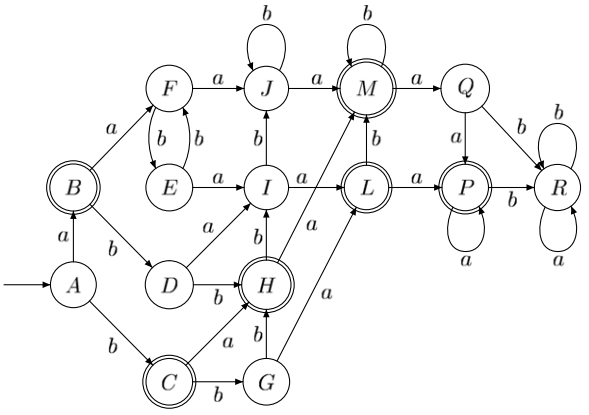

Fig. 1. The example dfa of [9, Figure 2].

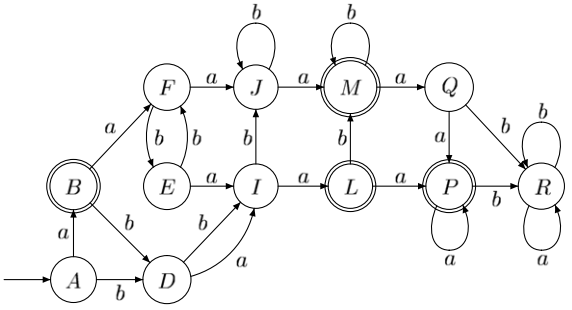

Fig. 2. The resulting hyper-minimal dfa for the input dfa of Fig. 1.

Algorithm 2 Merging almost-equivalent states (see [9]).

Require: a minimal dfa M, its kernel states K, and its almost-equivalent states $\sim$
Return: a hyper-minimal, almost-equivalent dfa

for all $B \in (Q/\sim)$ **do**
  if $B \cap K \neq \emptyset$ **then**
    select $q \in B \cap K$ // select a kernel state q from B, if one exists
  else
    select $q \in B$ // otherwise pick a preamble state of B
  for all $p \in B \setminus K$ **do**
    $(\delta, q_0) \leftarrow \text{merge}(\delta, q_0, p, q)$ // merge all preamble states of the block into q
return M

Example 3. Let us consider the minimal dfa of Fig. 1. Its kernel states are

$$\{E, F, I, J, L, M, P, Q, R\}.$$

It will be shown in Section 3.1, how to compute this set. The almost-equivalence $\sim$ is the equivalence relation induced by the partition

$$\{\{C, D\}, \{G, H, I, J\}, \{L, M\}, \{P, Q\}\},$$

which we show in Section 3.2. Now we enter the main loop of Algorithm 2.

- From the block $\{C, D\}$ we select the preamble state D. Thus, the state C is merged into D.
- From the block $\{G, H, I, J\}$, we select the kernel state I, and consequently, the states G and H are merged into I.
- In the blocks $\{L, M\}$ and $\{P, Q\}$ there are no preamble states to be merged.

The result of all merges is the dfa displayed in Fig. 2. It coincides with the dfa of [9, Figure 3].

Consequently, if we can also implement the procedures: (i) ComputeKernel and (ii) AEquivalentStates of Algorithm 1 in time $O(m \log n)$, then we obtain a hyper-minimization algorithm that runs in time $O(m \log n)$. The next two sections will show suitable implementations for both procedures.

Algorithm 3 Tarjan's algorithm computing the strongly connected components of M.

Require: a dfa $M = (Q, \Sigma, q_0, \delta, F)$ and a state $q \in Q$
Global: index, low: $Q \to \mathbb{N}$ initially undefined, $i = 0$, S stack of states initially empty

```
    index(q) ← i                                  // set index of q to i; q is now explored
 2: low(q) ← i                                    // set lowest index (of a state) reachable from q to the index of q
    i ← i + 1                                     // increase current index
 4: PUSH(S, q)                                    // push state q to the stack S
    for all σ ∈ Σ do
 6:   if index(δ(q, σ)) is undefined then
        Tarjan(M, δ(q, σ))                        // if successor not yet explored, then explore it
 8:     low(q) ← min(low(q), low(δ(q, σ)))        // update lowest reachable index for q
      else
10:     if δ(q, σ) ∈ S then
          low(q) ← min(low(q), index(δ(q, σ)))    // update lowest reachable index
12: if low(q) = index(q) then
      repeat
14:     p ← POP(S)                                // found component; remove all states of it from stack S
        ...                                       // store strongly connected components
16:   until p = q
```

3.1. Identification of kernel states

As we have seen in Algorithm 2, kernel states play a special rôle because we never merge two kernel states. It is shown in [9–11], how to identify the kernel states in time $O(mn)$. However, the kernel states can also be computed using a well-known algorithm (see Algorithm 3) due to Tarjan [16] in time $O(m)$.

Theorem 4. $\mathrm{Ker}(M)$ *can be computed in time* $O(m)$.

Proof. With Tarjan's algorithm [16] (or equivalently the algorithms by Gabow [17,18] or Kosaraju [19,20]) we can identify the strongly connected components (strongly connected states) in time $O(m+n)$. Algorithm 3 presents a simplified version of the general known algorithm because in our setting all states of M are reachable from q_0. The initial call is $Tarjan(M, q_0)$. At the same time, we also identify all center states because a center state is part of a strongly connected component of at least two states or has a self-loop; i.e., $\delta(q, \sigma) = q$ for some $\sigma \in \Sigma$. Another depth-first search can then mark all states q such that $p \to q$ for some center state p in time $O(m)$. Clearly, such a marked state is a kernel state and each kernel state $q \in \mathrm{Ker}(M)$ is marked because there exists a center state $p \in Q$ such that $p \to q$ by [9, Lemma 2.12]. □

Example 5. Let us again use the example dfa of Fig. 1. Tarjan's algorithm returns the set

$$\{\{A\}, \{B\}, \{C\}, \{D\}, \{E, F\}, \{G\}, \{H\}, \{I\}, \{J\}, \{L\}, \{M\}, \{P\}, \{Q\}, \{R\}\}$$

of strongly connected components. Consequently, the center states are $\{E, F, J, M, P, R\}$, and the depth-first search marks the states $\{E, F, I, J, L, M, P, Q, R\}$, which is the set of kernel states.

3.2. Identification of almost-equivalent states

The identification of almost-equivalent states is slightly more difficult. We improve the strategy of [9], which runs in time $O(mn^2)$,

- by avoiding pairwise comparisons, which yields an improvement by a factor n, and
- by merging states with a specific strategy, which reduces a factor n to $\log n$.

Essentially, the same strategy was independently employed by [14].

Let us attempt to explain Algorithm 4. The vector $(\delta(q, \sigma) \mid \sigma \in \Sigma)$ is called the *follow-vector* of q. Formally, the follow-vector is an element of Q^Σ, which denotes the set of all functions $f: \Sigma \to Q$. The algorithm keeps a set I of states that need to be processed and a set P of states that are still useful. Both sets are initially Q and the hash map h, which is of type $h: Q^\Sigma \to Q$, is initially empty; i.e., all values are unassociated. Moreover, the algorithm sets up a partition π of Q, which is initially the trivial partition, in which each state forms its own block (lines 1 and 2). The algorithm iteratively processes a state of I and computes its follow-vector. If the follow-vector is not yet associated in h, then the follow-vector will simply be stored in h. The algorithm proceeds in this fashion until it finds a state, whose follow-vector is already stored in h. It then extracts the state with the same follow-vector from h and compares the sizes of the blocks in π that the two states belong to. Suppose (without loss of generality) that p (q, respectively) is the state that belongs to the smaller (larger, respectively)

Algorithm 4 Algorithm computing $\sim$.

Require: minimal dfa $M = (Q, \Sigma, q_0, \delta, F)$
Return: the almost-equivalence relation $\sim$ represented as a partition

```
    for all q ∈ Q do
2:    π(q) ← {q}                                  // initial block of q contains just q itself
    h ← ∅                                         // hash map of type h: Q^Σ → Q
4:  I ← Q                                         // states that need to be considered
    P ← Q                                         // set of current states
6:  while I ≠ ∅ do
      q ← RemoveHead(I)                           // remove state from I
8:    succ ← ( δ(q, σ) | σ ∈ Σ )                  // compute vector of successors using current δ
      if HasValue(h, succ) then
10:     p ← Get(h, succ)                          // retrieve state in bucket succ of h
        if |π(p)| ≥ |π(q)| then
12:       Swap(p, q)                              // exchange rôles of p and q
        P ← P \ {p}                               // state p will be merged into q
14:     I ← I ∪ {r ∈ P | ∃σ: δ(r, σ) = p}         // add predecessors of p in P to I
        (δ, q0) ← merge(δ, q0, p, q)              // merge states p and q in δ; q survives
16:     π(q) ← π(q) ∪ π(p)                        // p and q are almost-equivalent
      h ← Put(h, succ, q)                         // store q in h under key succ
18: return π
```

block. Then we merge p into q and remove p from P because it is now useless. In addition, we update the block of q to include the block of p and add all states that have transitions leading to p to I because their follow-vectors have changed due to the merge. Note that the last step might add q to I again. The algorithm repeats this process until the set I is empty, which indicates that all states have been processed.

Let us proceed with an example run of Algorithm 4.

Example 6. Consider the minimal dfa of Fig. 1. Let us show the run of Algorithm 4 on it. We present a protocol (for line 10) in Table 1. At the end of the algorithm the hash map contains the following entries (we list the follow-vectors as vectors, in which the first and second component refer to a and b, respectively):

$$\binom{B}{C} \to A \quad \binom{F}{D} \to B \quad \binom{H}{G} \to C \quad \binom{I}{H} \to D \quad \binom{I}{F} \to E$$
$$\binom{J}{E} \to F \quad \binom{L}{H} \to G \quad \binom{M}{I} \to H \quad \binom{L}{J} \to I \quad \binom{M}{J} \to J$$
$$\binom{P}{M} \to L \quad \binom{Q}{M} \to M \quad \binom{P}{R} \to P \quad \binom{R}{R} \to R \quad \binom{L}{I} \to I$$
$$\binom{I}{E} \to F \quad \binom{I}{I} \to C \quad \binom{I}{G} \to E \quad \binom{F}{C} \to B.$$

From Table 1 we obtain the final partition

$$\{\{A\}, \{B\}, \{C, D\}, \{E\}, \{F\}, \{G, H, I, J\}, \{L, M\}, \{P, Q\}, \{R\}\}.$$

This coincides with the partition obtained in [9, Figure 2].

In the next sections we will take a detailed look at the time complexity (Section 4) and the correctness (Section 5) of Algorithm 4.

4. Time complexity of Algorithm 4

In this and the next section, we only discuss Algorithm 4, so all line references are to Algorithm 4 unless explicitly stated otherwise. Obviously, the hash map avoids the pairwise comparisons, and here we will show that our merging strategy realizes the reduction of a factor n to just $\log n$ (compared to the algorithm of [9]). Line 14 is particularly interesting for the time complexity because it might add to the set I, which controls the main loop. We start with a few simple loop invariants.

(i) $I \subseteq P$,
(ii) $\{\pi(p) \mid p \in P\}$ is a partition of Q,

Table 1
Run of Algorithm 4 (at line 10) on the automaton of Fig. 1.

I	$Q \setminus P$	q	p	π (singleton blocks not shown)
$\{B, \dots, R\}$	$\emptyset$	A		
$\dots$	$\emptyset$			
$\{R\}$	$\emptyset$	P	Q	
$\{M\}$	$\{Q\}$	R		$\{P, Q\}$
$\emptyset$	$\{Q\}$	M	L	$\{P, Q\}$
$\{H\}$	$\{M, Q\}$	J	I	$\{L, M\}, \{P, Q\}$
$\{F, I\}$	$\{J, M, Q\}$	H		$\{I, J\}, \{L, M\}, \{P, Q\}$
$\{I\}$	$\{J, M, Q\}$	F		$\{I, J\}, \{L, M\}, \{P, Q\}$
$\{C, D, G\}$	$\{J, M, Q\}$	I	H	$\{I, J\}, \{L, M\}, \{P, Q\}$
$\{D, G\}$	$\{H, J, M, Q\}$	C		$\{H, I, J\}, \{L, M\}, \{P, Q\}$
$\{G\}$	$\{H, J, M, Q\}$	D	C	$\{H, I, J\}, \{L, M\}, \{P, Q\}$
$\{B\}$	$\{D, H, J, M, Q\}$	G	I	$\{C, D\}, \{H, I, J\}, \{L, M\}, \{P, Q\}$
$\emptyset$	$\{D, G, H, J, M, Q\}$	B		$\{C, D\}, \{G, H, I, J\}, \{L, M\}, \{P, Q\}$

(iii) $(\delta(r, \sigma) \mid \sigma \in \Sigma) \in P^\Sigma$ for every $r \in Q$,
(iv) $h(P^\Sigma) = P \setminus I$, and
(v) $h^{-1}(P \setminus I) \cap P^\Sigma = \{(\delta(r, \sigma) \mid \sigma \in \Sigma) \mid r \in P \setminus I\}$.

Naturally, the symbols used refer to the ones of Algorithm 4 with their current values. Roughly speaking, (i) means that no useless state is ever active. The second statement yields that for every $q \in Q$ there exists an equivalent $p \in P$. The third and fourth statement essentially show that useless states have no incoming transitions and the follow-vectors that are stored in h belong to useful, but inactive states. Together, those statements guarantee that $p \neq q$ in lines 10–16. Finally, statements (iv) and (v) together say that the current follow-vectors of all useful, but inactive states (and only those) are stored in h and that they are all different.

Lemma 7. *Before every execution of line* 6 *we have:*

(i) $I \subseteq P$,
(ii) $\{\pi(p) \mid p \in P\}$ *is a partition of* Q,
(iii) $(\delta(r, \sigma) \mid \sigma \in \Sigma) \in P^\Sigma$ *for every* $r \in Q$,
(iv) $h(P^\Sigma) = P \setminus I$, *and*
(v) $h^{-1}(P \setminus I) \cap P^\Sigma = \{(\delta(r, \sigma) \mid \sigma \in \Sigma) \mid r \in P \setminus I\}$.

Proof. Clearly, we have to prove that the properties are true before entering the main loop and are preserved in each iteration. Trivially, all statements are true before entering the loop because $I = Q = P$, $h(Q^\Sigma) = \emptyset$, and each state is its own block (after execution of lines 1–2). In the loop, the state q is removed from I in line 7. Thus, $q \in P$ by statement (i). Next, its follow-vector succ is computed. Note that $q \in P \setminus I$ because I no longer contains q. Moreover, $q \notin h(P^\Sigma)$, which means that q has no association to a current follow-vector.

If no value is stored in h for succ, then succ is associated to q in h. Clearly, $I \subseteq P$, which proves statement (i). Statements (ii) and (iii) trivially remain true because neither P nor π nor δ are changed. Since $q \in P \setminus I$, succ $\in P^\Sigma$, and succ $\notin h^{-1}(Q)$, statements (iv) and (v) are also true, where for statement (v) we additionally use that q was not associated to a current follow-vector. This proves all statements in this case.

If the condition of line 9 is true, then the state p that is stored under the follow-vector succ is retrieved. Note that $p \in P \setminus I$ and $p \neq q$ by statements (iii) and (iv). Since we only know $q \in P \setminus I$ about q, the swap is irrelevant for the remainder of the proof. Without loss of generality, suppose that the condition in line 11 is false. If it is true and the swap occurs, then we can prove the statements in the same fashion with p and q exchanged. Next, p is removed from P in line 13 and all states of this new P that have a transition to p are added to I. Note that we might add q to I, but cannot add p to I. Since we only added states of P to I, we proved statement (i). Next, we merge p into q, which yields that all transitions to p are redirected to q. Since $p \neq q$ and $q \in P$, we proved statement (iii). In line 16 we combine the blocks of p and q in π. Statement (ii) is true because $p \notin P$, $q \in P$, and the result is clearly a partition. In the final step, we associate succ with q in h. For statements (iv) and (v), let us remark that $P \setminus I$, when compared to its value U in the previous iteration, now no longer contains p and every state added to I in line 14, but might now contain q if $q \notin I$. Each state of U had exactly one association to its then (before the merge in line 15) current follow-vector in h by statements (iv) and (v). If its follow-vector changed due to the merge, then it is no longer in $h(P^\Sigma)$ and no longer in $P \setminus I$ because the follow-vector changes if and only if it has a transition to p. If $q \notin I$, then succ $\in P^\Sigma$ is the current follow-vector of q and it replaces the entry for p. Thus, we obtain statements (iv) and (v). □

Now we are ready to state the main complexity lemma. To simplify the argument, we call (r, σ) a transition for every $r \in Q$ and $\sigma \in \Sigma$. Note that the value of $\delta(r, \sigma)$ might change in the course of the algorithm due to merges. For this reason, we did not include the target state in the transition (r, σ). Recall that $n = |Q|$.

Lemma 8. *For every* $r \in Q$ *and* $\sigma \in \Sigma$, *the transition* (r, σ) *is considered at most* $(\log n)$ *times in lines* 14 *and* 15 *during the full execution of Algorithm* 4.

Proof. Suppose that $p = \delta(r, \sigma)$ in line 14. Moreover, $|\pi(p)| < |\pi(q)|$ by lines 11–12. Then line 15 redirects the transition (r, σ) to q; i.e., $\delta(r, \sigma) = q$ after line 15. Moreover, $|\pi(q)| > 2 \cdot |\pi(p)|$ after the execution of line 16 because $p \neq q$ and $\pi(p) \cap \pi(q) = \emptyset$ by statements (ii)–(iv) of Lemma 7. Moreover, $|\pi(q)| \leq n$ for every $q \in Q$ by statement (ii) of Lemma 7. Consequently, (r, σ) can be considered at most $(\log n)$ times in lines 14 and 15, which proves the statement. □

Now we are ready to determine the run-time complexity of Algorithm 4. Recall that $m = |Q \times \Sigma|$ and $n = |Q|$. In addition, we exclude the nonsensical case $\Sigma = \emptyset$. Thus $m \geq n$. If we were to consider partial dfa, then we could set m to the number of existing transitions in a partial dfa. However, we continue to work with (total) dfa.

Theorem 9. *Algorithm 4 can be implemented to run in time $O(m \log n)$.*

Proof. Clearly, we assume that all basic operations except for those in lines 14 and 15 execute in constant time. Then lines 1–5 execute in time $O(n)$. Next we will prove that the loop in lines 6–17 executes at most $O(m \log n)$ times. By statement (i) of Lemma 7 we have $I \subseteq P$. Now let us consider a particular state $q \in Q$. Then $q \in I$ initially and it has $|\Sigma|$ outgoing transitions. By Lemma 8, every such transition is considered at most $(\log n)$ times in line 14, which yields that q is added to I. Consequently, the state q can be chosen in line 10 at most $(1 + |\Sigma| \cdot \log n)$ times. Summing over all states of Q, we obtain that the loop in lines 6–17 can be executed at most $(n + m \cdot \log n)$ times, which is in $O(m \log n)$ because $m \geq n$. Since all lines apart from lines 14 and 15 are assumed to execute in constant time, this proves the statement for all lines apart from 14 and 15. By Lemma 8 every transition is considered at most $(\log n)$ times in those two lines. Since there are m transitions and each consideration of a transition can be assumed to run in constant time, we obtain that lines 14 and 15 globally (i.e., including all executions of those lines) execute in time $O(m \log n)$, which proves the statement. □

To obtain a lower bound on the complexity, let us argue that minimization linearly reduces to hyper-minimization. Let M be a dfa that is not necessarily minimal. If $L(M) = \emptyset$, which can be verified in time $O(m)$, then we are done because the hyper-minimal dfa with one state that accepts the empty language is also minimal. Now let $L(M) \neq \emptyset$ and assume # to be a new input symbol not contained in Σ. We construct a dfa $M' = (Q, \Sigma \cup \{\#\}, q_0, \delta', F)$ by $\delta'(q, \sigma) = \delta(q, \sigma)$ for every $q \in Q$ and $\sigma \in \Sigma$ and $\delta'(q, \#) = q_0$ for every $q \in Q$. Observe, that by construction M' consists of kernel states only. Thus, hyper-minimizing M' leads to a dfa M'' that is unique because for two almost-equivalent hyper-minimized automata the kernels are isomorphic to each other [9, Theorem 3.5]. This should be compared with the characterization of minimal and hyper-minimal dfa mentioned in the Introduction. Thus, M'' is a minimal dfa accepting $L(M')$. Then it is easy to see that taking M'' and deleting the #-transitions yields a minimal dfa accepting $L(M)$. Hence, minimization linearly reduces to hyper-minimization. Thus, our algorithm achieves the optimal worst-case complexity in the light of the recent developments for Hopcroft's state minimization algorithm, which show that the $O(m \log n)$ bound is tight for that algorithm [7] even under any possible implementation [8].

5. Correctness of Algorithm 4

In this section, we prove that Algorithm 4 is correct. We will use [9, Lemma 2.10] for the correctness proof. To keep the paper self-contained, we repeat the required result and sketch its proof. Recall that $\sim$ is the almost-equivalence and that all congruences are relative to M.

Lemma 10 *([9, Lemma 2.10]). The equivalence $\sim$ is the most refined congruence $\simeq$ such that (†) for every $p, q \in Q$: $\delta(p, \sigma) \simeq \delta(q, \sigma)$ for every $\sigma \in \Sigma$ implies $p \simeq q$.*

Proof. Clearly, the congruences with property (†) are closed under intersection. Since there are only finitely many congruences, the most refined congruence $\simeq$ with property (†) exists. Moreover, $\sim$ is trivially a congruence [9, Lemma 2.9]. Thus, $\simeq \subseteq \sim$. For the converse, suppose that $p \sim q$. Then by Definition 1, there exists an integer $k \geq 0$ such that $\delta(p, w) = \delta(q, w)$ for all $w \in \Sigma^*$ with $|w| \geq k$. Trivially, $\delta(p, w) \simeq \delta(q, w)$ for all such words w, and for every $w' \in \Sigma^*$ if $\delta(p, w'\sigma) \simeq \delta(q, w'\sigma)$ for every $\sigma \in \Sigma$, then $\delta(p, w') \simeq \delta(q, w')$ by (†). Consequently, $p \simeq q$, which proves the statement. □

By statement (ii) of Lemma 7, $\{\pi(p) \mid p \in P\}$ is a partition of Q before every execution of line 6. Next, we prove that the induced equivalence relation is a congruence.

Lemma 11. *Before every execution of line 6, π induces a congruence $\simeq$ with $\simeq \subseteq \sim$.*

Proof. Let $\overline{\delta} = \delta$ be the transition function of M at the beginning of the algorithm. We prove the following loop invariants:

(i) $\simeq$ is a congruence,
(ii) $\delta(r, \sigma) \simeq \overline{\delta}(r, \sigma)$ for every $r \in Q$ and $\sigma \in \Sigma$, and
(iii) $p \simeq q$ implies $p \sim q$ for every $p, q \in Q$.

Before entering the main loop, π trivially induces the identity congruence, which also shows statement (iii). Moreover, $\delta(r,\sigma) \simeq \overline{\delta}(r,\sigma)$ for every $r \in Q$ and $\sigma \in \Sigma$ because $\delta = \delta'$. If the condition in line 9 is false, then the statements trivially remain true. Thus, let us consider lines 15 and 16 where δ and π are changed to δ' and π', respectively. Moreover, let $\simeq$ and $\cong$ be the equivalences corresponding to π and π', respectively. Finally, p and q are clearly such that $\delta(p,\sigma) = \delta(q,\sigma)$ for every $\sigma \in \Sigma$.

Let $q_1 \cong q_2$ and $\sigma \in \Sigma$. Note that $\simeq$ is more refined than $\cong$. In general,

$$\begin{aligned}\delta'(q_1,\sigma) &= \begin{cases} q & \text{if } \delta(q_1,\sigma) = p \\ \delta(q_1,\sigma) & \text{otherwise} \end{cases} \\ &\cong \delta(q_1,\sigma) \simeq \overline{\delta}(q_1,\sigma) \end{aligned} \tag{1}$$

because $p \cong q$ and by statement (ii). This proves statement (ii). For the remaining statements (i) and (iii), either $q_1 \simeq q_2$ or $q_1 \simeq p$ and $q \simeq q_2$. The third case, in which $q_1 \simeq q$ and $p \simeq q_2$ can be handled like the second case. Let us handle the first case, in which statement (iii) trivially holds. Moreover, using the analogue of (1) for q_2 and (1) itself, we obtain

$$\begin{aligned}\delta'(q_2,\sigma) &= \begin{cases} q & \text{if } \delta(q_2,\sigma) = p \\ \delta(q_2,\sigma) & \text{otherwise} \end{cases} \\ &\cong \delta(q_2,\sigma) \simeq \delta(q_1,\sigma) \cong \delta'(q_1,\sigma) \end{aligned}$$

using also the congruence property $\delta(q_2,\sigma) \simeq \delta(q_1,\sigma)$. This proves $\delta'(q_2,\sigma) \cong \delta'(q_1,\sigma)$ because $\simeq \subseteq \cong$.

In the second case, $q_1 \simeq p$ and $q \simeq q_2$. In the same way as in the first case, we obtain

$$\delta'(q_1,\sigma) \cong \delta'(p,\sigma) \quad q_1 \sim p \tag{2}$$

$$\delta'(q_2,\sigma) \cong \delta'(q,\sigma) \quad q_2 \sim q. \tag{3}$$

Since $\delta'(p,\sigma) = \delta'(q,\sigma)$ we obtain $\delta'(q_1,\sigma) \cong \delta'(q_2,\sigma)$, which proves statement (i). Moreover, $\overline{\delta}(p,\sigma) \simeq \delta(p,\sigma) = \delta(q,\sigma) \simeq \overline{\delta}(q,\sigma)$ by statement (ii), and thus, $\overline{\delta}(p,\sigma) \sim \overline{\delta}(q,\sigma)$ for every $\sigma \in \Sigma$ by statement (iii). The almost-equivalence has property (†) by Lemma 10, which yields $p \sim q$. Together with (2) and (3), we obtain $q_1 \sim q_2$, which completes the proof. □

This proves that we compute a congruence that is more refined than the almost-equivalence $\sim$. Thus, if we could show that the computed congruence also has property (†), then we compute $\sim$ by Lemma 10. This is achieved in the next theorem.

Theorem 12. *The partition returned by Algorithm 4 induces $\sim$.*

Proof. Before we can prove the theorem as already indicated, we need two auxiliary loop invariants. Let $\overline{\delta} = \delta$ at the beginning of the algorithm, and let $\simeq$ be the congruence induced by π. We prove the two invariants

(i) $q_1 \simeq q_2$ implies $q_1 = q_2$ for every $q_1, q_2 \in P$, and
(ii) for every $q_1, q_2 \in P \setminus I$: if $\delta(q_1,\sigma) = \delta(q_2,\sigma)$ for every $\sigma \in \Sigma$, then $q_1 = q_2$.

Clearly, both statements are true before entering the loop because $\simeq$ is the equality and $P = Q = I$. If the condition in line 9 is false, then statement (i) trivially remains true. Since q is no longer in I, we need to prove statement (ii) for $q \in \{q_1, q_2\}$. Because there was no entry at succ in h, succ $\in P^\Sigma$, and $h(P^\Sigma) = P \setminus I$ by statements (iii) and (iv) of Lemma 7, we know that $q_1 = q = q_2$, which proves statement (ii).

Now when π is changed in line 16, we already merged p into q in line 15. Let $\cong$ be the equivalence induced by the new partition (after execution of line 17) and δ' be the transition function after the potential merge. Moreover, let $q_1, q_2 \in P$ such that $q_1 \cong q_2$. Note that $q_1 \neq p \neq q_2$ because $p \notin P$. As in the proof of Lemma 10, either $q_1 \simeq q_2$ or $q_1 \simeq p$ and $q \simeq q_2$. The third case is again symmetric to the second. The second case is contradictory because $q_1 \simeq p$ implies $q_1 = p$ by statement (i), but $q_1 \neq p$. Thus, $q_1 \simeq q_2$ and $q_1 = q_2$ by statement (i). For statement (ii), additionally, let $q_1, q_2 \in P \setminus I$ such that $\delta'(q_1,\sigma) = \delta'(q_2,\sigma)$ for every $\sigma \in \Sigma$. Then

$$\delta'(q_1,\sigma) = \begin{cases} q & \text{if } \delta(q_1,\sigma) = p \\ \delta(q_1,\sigma) & \text{otherwise} \end{cases}$$

$$\delta'(q_2,\sigma) = \begin{cases} q & \text{if } \delta(q_2,\sigma) = p \\ \delta(q_2,\sigma) & \text{otherwise.} \end{cases}$$

However, if the first case applies, then $q_1 \in I$ ($q_2 \in I$, respectively), which is contradictory. Thus, $\delta(q_1,\sigma) = \delta'(q_1,\sigma) = \delta'(q_2,\sigma) = \delta(q_2,\sigma)$, and we can use statement (ii) to prove the statement unless $q \in \{q_1, q_2\}$. Without loss of generality, let $q_1 = q$. Only the state p extracted in line 10 has the same follow-vector by statement (v) of Lemma 7, but $q_2 \neq p$. This proves $q_1 = q = q_2$, and thus, we proved the auxiliary statements.

Let $\simeq$ be the equivalence returned by Algorithm 4. By Lemma 11, the congruence $\simeq$ is more refined than the almost-equivalence $\sim$. Thus, if $\simeq$ has property (†), then $\simeq$ and $\sim$ coincide by Lemma 10. It remains to prove property (†) for $\simeq$. Let

$q_1, q_2 \in Q$ be such that $\overline{\delta}(q_1, \sigma) \simeq \overline{\delta}(q_2, \sigma)$ for every $\sigma \in \Sigma$. By assumption, statement (ii) of Lemma 7, and statements (i) and (ii) in the proof of Lemma 11, there exist $p_1 \simeq q_1$ and $p_2 \simeq q_2$ such that

$$\delta(p_1, \sigma) \simeq \delta(q_1, \sigma) \simeq \overline{\delta}(q_1, \sigma) \simeq \overline{\delta}(q_2, \sigma) \simeq \delta(q_2, \sigma) \simeq \delta(p_2, \sigma).$$

Due to statement (iii) of Lemma 7 we have $\delta(p_1, \sigma) \in P$ and $\delta(p_2, \sigma) \in P$. With the help of the first property we obtain $\delta(p_1, \sigma) = \delta(p_2, \sigma)$ for every $\sigma \in \Sigma$. Since the algorithm terminates with $I = \emptyset$, we can apply statement (ii) to obtain $p_1 = p_2$, which together with $q_1 \simeq p_1 = p_2 \simeq q_2$ proves that $q_1 \simeq q_2$. Thus, $\simeq$ has property (†). □

Finally, we can collect our results in the next theorem, which is the main contribution of this paper.

Theorem 13. *For every dfa we can obtain an almost-equivalent, hyper-minimal dfa in time $O(m \log n)$.*

6. Conclusions

We have designed an $O(m \log n)$ algorithm, where $m = |Q \times \Sigma|$ and $n = |Q|$, that computes a hyper-minimized dfa from a given dfa $(Q, \Sigma, q_0, \delta, F)$. The hyper-minimized dfa may have fewer states than the classical minimized dfa. Its accepted language is almost-equal to the original one, which means that it differs in acceptance on only a finite number of inputs. Since hyper-minimization is a very new field of research, most of the standard questions related to descriptional complexity such as, for example, *nondeterministic automata to dfa conversion* with respect to hyper-minimality, are problems of further research.

Acknowledgements

We would like to thank the reviewers of the conference and journal draft version for their helpful comments. In addition, we would like to express our gratitude to Artur Jeż for contacting us with his research and general discussion. Part of this work was done while the first author was at Institut für Informatik, Technische Universität München, Boltzmannstraße 3, D-85748 Garching bei München, Germany. The second author was supported by the *Ministerio de Educación y Ciencia* (MEC) grant JDCI-2007-760.

References

[1] M.O. Rabin, D. Scott, Finite automata and their decision problems, IBM J. Res. Dev. 3 (2) (1959) 114–125.
[2] A.R. Meyer, M.J. Fischer, Economy of description by automata, grammars, and formal systems, in: 12th Annual Symposium on Switching and Automata Theory, IEEE Computer Society, 1971, pp. 188–191.
[3] F.R. Moore, On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata, IEEE Trans. Comput. 20 (10) (1971) 1211–1214.
[4] T. Jiang, B. Ravikumar, Minimal NFA problems are hard, SIAM J. Comput. 22 (6) (1993) 1117–1141.
[5] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison Wesley, 1979.
[6] J.E. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, in: Theory of Machines and Computations, Academic Press, 1971, pp. 189–196.
[7] J. Berstel, O. Caston, On the complexity of Hopcroft's state minimization algorithm, in: Proc. 9th Int. Conf. Implementation and Application of Automata, in: LNCS, vol. 3317, Springer, 2004, pp. 35–44.
[8] G. Castiglione, A. Restivo, M. Sciortino, Hopcroft's algorithm and cyclic automata, in: Proc. 2nd Int. Conf. Languages, Automata Theory and Applications, in: LNCS, vol. 5196, Springer, 2008, pp. 172–183.
[9] A. Badr, V. Geffert, I. Shipman, Hyper-minimizing minimized deterministic finite state automata, RAIRO Theor. Inf. Appl. 43 (1) (2009) 69–94.
[10] A. Badr, Hyper-minimization in $O(n^2)$, in: Proc. 13th Int. Conf. Implementation and Application of Automata, in: LNCS, vol. 5148, Springer, 2008, pp. 223–231.
[11] A. Badr, Hyper-minimization in $O(n^2)$, Internat. J. Found. Comput. Sci. 20 (4) (2009) 735–746.
[12] C. Câmpeanu, N. Santean, S. Yu, Minimal cover-automata for finite languages, Theoret. Comput. Sci. 267 (1–2) (2001) 3–16.
[13] A. Paun, M. Paun, A. Rodríguez-Patón, On the Hopcroft's minimization technique for DFA and DFCA, Theoret. Comput. Sci. 410 (24–25) (2009) 2424–2430.
[14] P. Gawrychowski, A. Jeż, Hyper-minimisation made efficient, in: Proc. 34th Int. Symp. Mathematical Foundations of Computer Science, in: LNCS, vol. 5734, Springer, 2009, pp. 356–368.
[15] M. Holzer, A. Maletti, An $n \log n$ algorithm for hyper-minimizing states in a (minimized) deterministic automaton, in: Proc. 14th Int. Conf. Implementation and Application of Automata, in: LNCS, vol. 5642, Springer, 2009, pp. 4–13.
[16] R.E. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.
[17] J. Cheriyan, K. Mehlhorn, Algorithms for dense graphs and networks on the random access computer, Algorithmica 15 (6) (1996) 521–549.
[18] H.N. Gabow, Path-based depth-first search for strong and biconnected components, Inf. Process. Lett. 74 (3–4) (2000) 107–114.
[19] S.R. Kosaraju, Strong-connectivity algorithm, unpublished manuscript, 1978.
[20] M. Sharir, A strong-connectivity algorithm and its applications in data flow analysis, Comput. Math. Appl. 7 (1) (1981) 67–72.