An $n \log n$ Algorithm for Hyper-Minimizing States in a (Minimized) Deterministic Automaton

Markus Holzer[1,⋆] and Andreas Maletti[2,⋆⋆]

[1] Institut für Informatik, Universität Giessen
Arndtstr. 2, 35392 Giessen, Germany
holzer@informatik.uni-giessen.de

[2] Departament de Filologies Romàniques, Universitat Rovira i Virgili
Av. Catalunya 35, 43002 Tarragona, Spain
andreas.maletti@urv.cat

Abstract. We improve a recent result [A. Badr: Hyper-Minimization in $O(n^2)$. In *Proc. CIAA*, LNCS 5148, 2008] for hyper-minimized finite automata. Namely, we present an $O(n \log n)$ algorithm that computes for a given finite deterministic 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] 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.

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 w.r.t. the number of states [2, 3]. 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

⋆ Part of the work was done while the author was at Institut für Informatik, Technische Universität München, Boltzmannstraße 3, D-85748 Garching bei München, Germany.
⋆⋆ Supported by the *Ministerio de Educación y Ciencia* (MEC) grant JDCI-2007-760.

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 machine and collapsing states that are equivalent. Here two states $p, q \in Q$, where Q is the set of states of the automaton 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 a 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 allows one to give 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 the relation starting with a partition that groups accepting and rejecting states together yielding a polynomial time algorithm of $O(n^2)$; compare with [5]. HOPCROFT's algorithm [6] for minimization slightly improves the naive implementation to a running time of $O(m \log n)$ where $m = |Q \times \Sigma|$ and $n = |Q|$, where Σ is alphabet of input symbols of the finite automaton, and is up to now the best known minimization algorithm. 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, 10]. 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 [11, 12]. Hyperminimization [9, 10] 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-minimized 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 $O(m \cdot n^3)$, where $m = |\Sigma|$ and $n = |Q|$; for constant alphabet size this gives an $O(n^3)$ algorithm. Later, the bound was improved to $O(n^2)$ in [10]. In this paper we improve this upper bound further to $O(n \log n)$, and argue that it is reasonably well because any upper bound $t(n) = \Omega(n)$ for hyper-minimization implies that minimization can be done within $t(n)$. To this end, we linearly reduce minimization to hyper-minimization.

2 Preliminaries

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. A finite set Σ is an alphabet. By Σ^* we denote the set of all strings over Σ. The empty string is denoted by ε. Concatenation of strings is denoted by juxtaposition and $|w|$ denotes the length of the word $w \in \Sigma^*$. A deterministic finite automaton (dfa) is a tuple $M = (Q, \Sigma, q_0, \delta, F)$ where Q is a finite set of states, Σ is an alphabet of input symbols, $q_0 \in Q$ is the initial state, $\delta \colon Q \times \Sigma \to Q$ is a transition function, and $F \subseteq Q$ is a 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 $L(M) = \{ w \in \Sigma^* \mid \delta(q_0, w) \in F \}$.

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 $w \in \Sigma^*$. The dfa M is minimal if it does not have equivalent states. The name 'minimal' stems from the fact that no dfa with less states also recognizes $L(M)$ if M is minimal. It is known that for M an equivalent minimal dfa can efficiently be computed using Hopcroft's algorithm [6], which runs in time $O(m \log n)$ where $m = |Q \times \Sigma|$ and $n = |Q|$.

In the following, let $M = (Q, \Sigma, q_0, \delta, F)$ be a minimal dfa. Let us recall some notions from [9]. 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 states by $\mathrm{Ker}(M)$ and the set of preamble states by $\mathrm{Pre}(M)$. For states $p, q \in Q$ we write $p \to q$ if there exists $w \in \Sigma^+$ such that $\delta(p, w) = q$. The states p and q are strongly connected, denoted by $p \leftrightarrow q$, if $p \to q$ and $q \to p$. Note that strongly connected states are also a kernel states since both are reachable by the minimality of M. Finally, $q \in Q$ is a center state if $q \leftrightarrow q$.

3 Hyper-minimization

As already remarked, minimization yields an equivalent dfa that is as small as possible. It can thus be considered 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: We allow a finite number of errors. Consequently, we call two dfa M_1 and M_2 *almost-equivalent* if $L(M_1)$ and $L(M_2)$ are almost-equal. A dfa that admits no smaller almost-equivalent dfa is called *hyper-minimal*. Hyper-minimization [9, 10] aims to find an almost-equivalent hyper-minimal dfa.

Algorithm 1 Overall structure of the hyper-minimization algorithm.

Require: a dfa M

```
   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
```

The contributions [9, 10] report hyper-minimization algorithms for M that run in time $O(n^3)$ and $O(n^2)$, respectively. Note that $|\Sigma|$ is assumed to be constant in those contributions. Our aim here is to develop a hyper-minimization algorithm that runs in time $O(n \log n)$ under the same assumptions.

Roughly speaking, minimization aims to identify equivalent states and hyper-minimization aims to identify almost-equivalent states, which we define next. Recall that $M = (Q, \Sigma, q_0, \delta, F)$ is a minimal dfa. Let $m = |Q \times \Sigma|$ and $n = |Q|$.

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 $|w| \geq k$.*

Let us present the overall structure of the hyper-minimization algorithm of [10] in Algorithm 1. Note that compared to [10], we exchanged lines 2 and 3. Minimize refers to classical minimization. Hopcroft's algorithm implements it and runs in time $O(m \log n)$ [6]. The procedure MergeStates is described in [9, 10], where it is also proved that it runs in time $O(m)$. To make the paper self-contained, we present their algorithm (see Algorithm 2) and the corresponding results next. Note that merging a state p into another state q denotes the usual procedure of redirecting (in M) all incoming transitions of p to q. If p was the initial state, then q is the new initial state. Clearly, the state p can be deleted.

Theorem 2 ([9, Section 4]). *If the requirements of Algorithm 2 are met, then it returns in time $O(m)$ a hyper-minimal dfa that is almost-equivalent to M.*

Consequently, if we can implement: (i) ComputeKernel and (ii) AEquivalentStates 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.

3.1 Identify kernel states

As we have seen in Algorithm 2, kernel states play a special role because we never merge two kernel states. It was already shown in [9, 10], how to identify the kernel states in time $O(mn)$. It turns out that the kernel states can easily be computed using a well-known algorithm due to Tarjan [13] (see Algorithm 3).

Theorem 3. $\mathrm{Ker}(M)$ *can be computed in time $O(m)$.*

Algorithm 2 Merging almost-equivalent states.

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

```
    for all B ∈ (Q/∼) do
2:    S ← B ∩ K                          // S contains the kernel states of the block B
      if S ≠ ∅ then
4:      select q ∈ S                      // select an arbitrary kernel state q from B
      else
6:      select q ∈ B       // if no such kernel state exists, pick any state q of B
      for all p ∈ B \ S do
8:      merge p into q               // merge all preamble states of the block into q
    return M
```

Proof. Using TARJAN's algorithm [13] (or the algorithms by GABOW [14, 15] or KOSARAJU [16, 17]) we can identify the strongly connected components in time $O(m+n)$. Algorithm 3 presents a simplified formulation because all states of M are reachable from q_0. The initial call is TARJAN(M, q_0). Thus, we identified states q such that $q \to q$ because such a 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 $p \to p \to q$ for some state p in time $O(m)$. Clearly, such a marked state is a kernel state and each kernel state is marked because for each $q \in \mathrm{Ker}(M)$ there exists a state $p \in Q$ such that $p \to p \to q$ by [9, Lemma 2.12]. □

3.2 Identify almost-equivalent states

The identification of almost-equivalent states will be slightly more difficult. We improve the strategy of [9], which runs in time $O(mn^2)$, by avoiding pairwise comparisons, which yields a factor n, and by merging states with a specific strategy, which reduces a factor n to $\log n$. Since M is a minimal dfa, the relation $\sim$ coincides with the relation defined in [9, Definition 2.2]. Thus, we know that $\sim$ is a congruence relation by [9, Facts 2.5–2.7].

Let us attempt to explain the algorithm. The vector $\langle\, \delta(q, \sigma) \mid \sigma \in \Sigma \,\rangle$ is called the *follow-vector* of 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 is initially empty. The algorithm then iteratively processes states of I and computes their follow-vector. Since h is initially empty, the first 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 vector from h and compares the sizes of the blocks in π that the two states belong to. Suppose that p is the state that belongs to the smaller block and q is the state that belongs to the larger 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. The algorithm repeats this process until the set I is empty.

Algorithm 3 TARJAN's algorithm TARJAN(M, q) 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, $i \in \mathbb{N}$ initially 0, S stack of states initially empty

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

Example 4. Consider the minimal dfa of Figure 1(left) (see [9, Figure 2]). Let us show the run of Algorithm 4 on it. We present a protocol (for line 10) in Table 1. At then end of the algorithm the hash map contains the following entries:

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

From Table 1 we obtain the partition induced by $\sim$, which is

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

This coincides with the partition obtained in [9, Figure 2]. Since E, F, I, J, L, M, P, Q, and R are kernel states, we can only merge C into D and merge G and H into I. The result of those merges is shown in Figure 1(right). The obtained dfa coincides with the one of [9, Figure 3].

Next, let us look at the time complexity before we turn to correctness. In this respect, line 14 is particularly interesting because it might add to the set I, which controls the loop. Our strategy that determines which states to merge will realize the reduction of a factor n to just $\log n$. To simplify the argument, we

Algorithm 4 Algorithm computing $\sim$.

Require: minimal dfa $M = (Q, \Sigma, q_0, \delta, F)$

```
    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 roles of p and q
        P ← P \ {p}                            // state p will be merged into q
14:     I ← (I \ {p}) ∪ { r ∈ P | ∃σ: δ(r, σ) = p }  // add predecessors of p in P to I
        δ ← MergeState(δ, 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
```

will call $\delta(q, \sigma)$ a transition and we consider it the same transition even if the value of $\delta(q, \sigma)$ changes in the course of the algorithm.

Proposition 5. *The following properties of Algorithm 4 hold whenever line 7 is executed: (i) $I \subseteq P$ and (ii) $\{ \pi(p) \mid p \in P \}$ is a partition of Q.*

Moreover, let us consider p and q after the execution of line 10. In essence, we would like to show that $p \neq q$. We thus need to show that $(\delta(q, \sigma) \mid \sigma \in \Sigma) \neq \alpha$ for every $\alpha \in h^{-1}(q)$ whenever line 8 is executed. Clearly, when line 8 is first executed with our particular q, then $h^{-1}(q) = \emptyset$ and thus the property trivially holds. Moreover, q is then no longer in I. It can be added to I in line 14, but only if $\delta(q, \sigma) \notin P$ for some $\sigma \in \Sigma$. Then it is changed in line 15 such that $\delta(q, \sigma) \in P$ after its execution. Thus, all stored values $h^{-1}(q)$ have at least one component that is not in P, whereas $\delta(q, \sigma) \in P$ for every $\sigma \in \Sigma$ after execution of line 15. Consequently, in line 10 the retrieved state p cannot be q itself.

Lemma 6. *For every $r \in Q$ and $\sigma \in \Sigma$, the transition $\delta(r, \sigma)$ 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 $\delta(r, \sigma)$ 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 $p \neq q$ as already argued, and thus, $\pi(p) \cap \pi(q) = \emptyset$ by Proposition 5. Moreover, by the same proposition $|\pi(q)| \leq n$ for every $q \in Q$. Consequently, $\delta(r, \sigma)$ can be considered at most $(\log n)$ times in lines 14 and 15. □

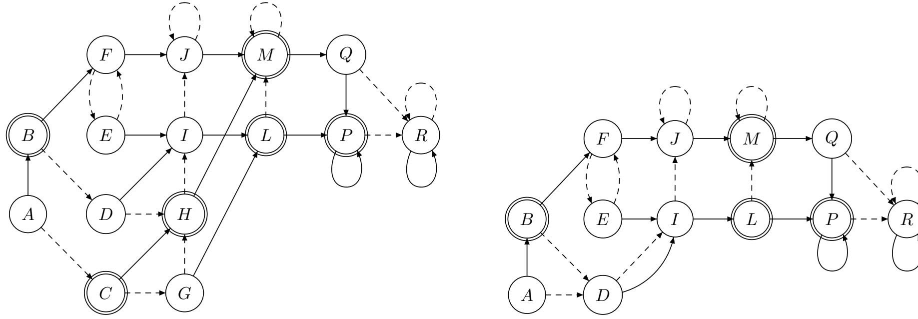

Fig. 1. An example automaton and the resulting hyper-minimal automaton with a-transitions (straight lines) and b-transitions (dashed lines). The initial state is A.

Theorem 7. *Algorithm 4 can be implemented to run in time $O(m \log n)$.*

Proof. Clearly, we assume that all operations except for those in lines 14 and 15 to 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 $O(m \cdot \log n)$ times. By Proposition 5 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 6, 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. 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 6 every transition is considered at most $(\log n)$ times in those two lines. Since there are m transitions in M 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. ◻

Finally, we need to prove that Algorithm 4 is correct. By Proposition 5, $\{\, \pi(p) \mid p \in P \,\}$ is a partition of Q whenever line 7 is executed. Let $\simeq$ be the induced equivalence relation. Next we prove that $\simeq$ is a congruence.

Lemma 8. *Whenever line 7 is executed, π induces a congruence.*

This proves that we compute a congruence. Now we can use [9, Lemma 2.10] to prove that all states in a block of the returned partition are almost-equivalent.

Theorem 9. *The partition returned by Algorithm 4 induces $\sim$.*

Proof. Let $\simeq$ be the congruence (see Lemma 8) returned by Algorithm 4. For every $\sigma \in \Sigma$ and $p, q \in Q$ that are merged in line 15 we have $\delta(p, \sigma) \sim \delta(q, \sigma)$. Thus, $p \sim q$ by [9, Lemma 2.10], which proves $\simeq \subseteq \sim$. For the converse, let $p \sim q$. Clearly, δ is the transition function of $M/\!\simeq$ at the end of the algorithm. Denote

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

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

the transition function of $M/{\simeq}$ by δ' and the original transition function of M by δ. Since $p \sim q$, there exists $k \geq 0$ such that $\delta(p,w) = \delta(q,w)$ for every $w \in \Sigma^*$ with $|w| \geq k$. Clearly, this yields that $\delta'([p],w) = \delta'([q],w)$ for every such w. This implies the existence of $B, D \in (Q/{\simeq})$ such that $\delta'(B,\sigma) = \delta'(D,\sigma)$ for every $\sigma \in \Sigma$. However, an easy proof shows that the algorithm does not terminate as long as there are distinct states B and D such that $\delta'(B,\sigma) = \delta'(D,\sigma)$ for every $\sigma \in \Sigma$. Consequently, $p \simeq q$, which proves the statement. □

Theorem 10. *For every dfa we can obtain a almost-equivalent, hyper-minimal dfa in time $O(m \log n)$.*

4 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, which may have fewer states than the classical minimized dfa. Its accepted language is almost-equivalent to the original one; i.e., differs in acceptance on a finite number of inputs only. Since hyper-minimization is a very new field of research, most of the standard questions related to descriptional complexity such as, e.g., nondeterministic automata to dfa conversion w.r.t. hyper-minimality, are problems of further research.

Finally, let's argue that minimization linearly reduces to hyper-minimization. This is seen as follows: Let $M = (Q, \Sigma, q_0, \delta, F)$ be a dfa. If $L(M) = \emptyset$, which can be verified in time linear in the number of states, then we are already done since the single state hyper-minimal dfa accepting the emptyset 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 $\delta'(p,\sigma) = \delta(p,\sigma)$ for $p \in Q$ and $\sigma \in \Sigma$ and $\delta'(p,\#) = q_0$ for $p \in Q$. 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]—compare this 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 $O(n \log n)$ bound is tight for that algorithm [7] even under any possible implementation [8].

References

1. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM J. Res. Dev. **3** (1959) 114–125
2. Meyer, A.R., Fischer, M.J.: Economy of description by automata, grammars, and formal systems. In: Annual Symposium on Switching and Automata Theory, IEEE Computer Society Press (1971) 188–191
3. Moore, F.R.: On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata. IEEE Transaction on Computing **C-20** (1971) 1211–1219
4. Jiang, T., Ravikumar, B.: Minimal NFA problems are hard. SIAM J. Comput. **22**(6) (1993) 1117–1141
5. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley (1979)
6. Hopcroft, J.E.: An $n \log n$ algorithm for minimizing states in a finite automaton. In: Theory of Machines and Computations. Academic Press (1971) 189–196
7. Berstel, J., Caston, O.: On the complexity of Hopcroft's state minimization algorithm. In: Conference on Implementation and Application of Automata (CIAA). Number 3317 in LNCS, Springer (2004) 35–44
8. Castiglione, G., Restivo, A., Sciotino, M.: Hopcroft's algorithm and cyclic automata. In: Conference on Languages, Automata Theory and Applications (LATA). Number 5196 in LNCS, Springer (2008) 172–183
9. Badr, A., Geffert, V., Shipman, I.: Hyper-minimizing minimized deterministic finite state automata. RAIRO Theor. Inf. Appl. **43**(1) (2009) 69–94
10. Badr, A.: Hyper-minimization in $O(n^2)$. In: Conference on Implementation and Application of Automata (CIAA). Number 5148 in LNCS, Springer (2008) 223–231
11. Câmpeanu, C., Santean, N., Yu, S.: Minimal cover-automata for finite languages. Theoret. Comput. Sci. **267**(1–2) (2001) 3–16
12. Paun, A., Paun, M., Rodríguez-Patón, A.: On the Hopcroft minimization technique for DFA and DCFA. Theoret. Comput. Sci. (2009) to appear. available at: `http://dx.doi.org/10.1016/j.tcs.2009.02.034`.
13. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2) (1972) 146–160
14. Cheriyan, J., Mehlhorn, K.: Algorithms for dense graphs and networks. Algorithmica **15**(6) (1996) 521–549
15. Gabow, H.N.: Path-based depth-first search for strong and biconnected components. Inf. Process. Lett. **74**(3–4) (2000) 107–114
16. Kosaraju, S.R.: Strong-connectivity algorithm. unpublished manuscript (1978)
17. Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. Computers and Mathematics with Applications **7**(1) (1981) 67–72