A split-based incremental deterministic automata minimization algorithm

Pedro Garcia $^{(1)}$, Manuel Vázquez de Parga$^{(1)}$, Jairo A. Velasco$^{(2)}$ and Damián López$^{(1)}$

$^{(1)}$ Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
{pgarcia,mvazquez,dlopez}@dsic.upv.es

$^{(2)}$ Pontificia Universidad Javeriana. Cali (Colombia)
jairov@javerianacali.edu.co

Abstract

We here study previous results due to Hopcroft and Almeida et al. to propose an incremental split-based deterministic automata minimization algorithm whose average running-time does not depend on the size of the alphabet. The experimentation carried out shows that our proposal outperforms the algorithms studied whenever the automata have more than a (quite small) number of states and symbols.

Keywords: Finite automata; DFA *minimization; incremental minimization*

1 Introduction

The algorithms that explicitly compute the Nerode's equivalence to minimize deterministic finite automata (*DFA*) can be classified into two main categories. The first one takes into account every pair of states in the automaton and search for a string to distinguish both states. Algorithms that follow this approach are due to Huffmann-Moore [1], Watson and Daciuk [2], Almeida et al. [3] among others. The second approach considers an initial partition of the automaton set of states into final and non-final states. This partition is subsequently refined in order to obtain the partition induced by the Nerode's equivalence relation. The algorithms by Hopcroft [4] and Moore [5] follow this strategy. See [6] for a review of these methods.

It is well known that the most time-efficient method is the one by Hopcroft. This method has a worst-case running-time $\mathcal{O}(kn \log n)$ where n denotes the number of states of the automaton and k the number of symbols in the alphabet. It has also been stablished that the average time-complexity of Moore's and Hopcroft's algorithms is $\mathcal{O}(kn \log \log n)$ [7].

Recently, Almeida et al. propose an incremental algorithm with time complexity of $\mathcal{O}(kn^2\alpha(n))$, where $\alpha(n)$ is a function related to the inverse of Ackermann's function, that grows so slowly that can be taken as constant. The experiments carried out by the authors using a corpus of uniformly random-generated automata show that, despite the algorithm is linear with respect the size of the alphabet, the average running-time of the algorithm does not depend on the size of the alphabet. This is due to the fact that, taking into account automata from a uniform distribution, once the number of states and symbols in the alphabet is over a (quite small) bound, almost every generated automaton is already minimal [8].

The experimental behavior of the incremental algorithm proposed by Almeida et al. is interesting in some circumstances where the number of symbols is comparable to the number of states because, in those cases, this algorithm theoretically outperforms Hopcroft's method.

In this paper we propose an incremental split-based minimization algorithm whose averaged running-time does not depend on the size of the alphabet. We used the automata dataset used in [3] in an experimentation that confirms that, in certain circumstances (automata with big enough alphabet), our proposal outperforms the algorithms by Hopcroft and Almeida et al.

2 Preliminaries

2.1 Notation and definitions

We recall in this section the essential definitions for the results presented. We refer the interested reader to the book by Hopcroft and Ullman [1].

Let Σ be a finite alphabet and let Σ^* (the set of all possible words using symbols in Σ) be the free monoid generated by Σ with concatenation as the internal operation and the empty string λ as neutral element. A language L over Σ is defined as any subset of Σ^*.

A *deterministic finite automaton* (*DFA*) is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is an alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and $\delta : Q \times \Sigma \to Q$ is the transition function. When the transition function is a total function then the automaton is said to be *complete*. The transition function can also be seen as $\delta \subseteq (Q \times \Sigma \times Q)$, and can be extended in a natural way to Σ^*. We say the *DFA* A is accessible if, for each $q \in Q$, there exists a string x such that $\delta(q_0, x) = q$. In the following we will assume that *DFA*s are complete and accessible.

The *right language* of a state q of a *DFA* A is defined as $R_q^A = \{x \in \Sigma^* : \delta(q, x) \in F\}$. Two states p and q of an automaton A are said to be equivalent when $R_p^A = R_q^A$. A *DFA* is minimal when it has no two equivalent states. The minimal *DFA* for a given language is unique up to isomorphism, and any process to obtain this *DFA* for a language is equivalent to the computation of the Nerode's equivalence for the language [1].

A *partition* π of a set Q is a set $\{P_1, P_2, \ldots, P_k\}$ of pairwise disjoint non-empty subsets of Q such that the union of all the P_i equals Q. We will refer to the subsets of a partition as *blocks*, and we will denote the block of π which contains p with $[p]_\pi$. A partition π_1 is refined by π_2 (π_1 is coarser than π_2) if each class in π_2 is contained in some class in π_1. Let π_1 and π_2 be two partitions of a set Q, we will denote with $\pi_1 \wedge \pi_2$ the coarsest partition which refines both π_1 and π_2. The classes of this partition are the non empty sets in $P_1 \cap P_2$, where $P_1 \in \pi_1$ and $P_2 \in \pi_2$.

Given a *DFA* $A = (Q, \Sigma, \delta, q_0, F)$, let $P, R \subset Q$ and $a \in \Sigma$. Let us refer to (P, a) as a *splitter* and also denote by $(P, a)|R$ the *split* of the set R into the sets $R' = \delta^{-1}(P, a) \cap R$ and $R'' = R - R'$ when they are non-empty.

2.2 *DFA* minimization algorithms

Hopcroft's algorithm has the best theoretical time-complexity bound to minimize automata [4]. Many papers are devoted to describe this method [9, 10, 11, 12, 6] and it is outlined in Algorithm 2.1. Briefly speaking, the algorithm maintains a list of splitters (a pair containing a symbol and a subset of the set of states) that allow to successively refine the initial partition of the set of states into final and non-final states. A careful implementation of this algorithm leads to a worst-case time-complexity $\mathcal{O}(kn \log n)$.

Despite its worse theoretical time-complexity bound, an algorithm that in some circumstances performs better than Hopcroft's is due to Almeida et al. [3]. This algorithm is presented as an evolution of a previous algorithm by Watson and Daciuk [2] and follows a similar approach to the one by Huffmann and Moore described in [1]. Instead of refining an initial partition of the set of states, this incremental algorithm considers the sets of final and non-final states to establish the first pairs of non-equivalent states. Then, using the transition function of the automaton and for each pair (p, q) of states whose relationship has not yet been stablished, the algorithm carries out a recursive depth-first search for equivalence (or non-equivalence) evidence. In order to avoid extra computation, for any pair of states, a normalization function orders the states in the pair and outputs the result. When this traversal concludes that the states are equivalent, each pair of explored states is also annotated as equivalent. When the traversal finds a pair (p', q') of non-equivalent states, then each pair in the path from (p, q) to (p', q') is annotated as non-equivalent. Algorithms 2.2 and 2.3 describe the method in a similar way the authors do in their paper. We here do not intend to fully describe the algorithm by Almeida et al. and refer the interested reader to [3] for further information (including implementation details).

As mentioned in the introduction, this incremental algorithm has a the-

Algorithm 2.1 Hopcroft's *DFA* minimization algorithm.

```
Input: A DFA $A$
Output: The minimal DFA equivalent to $A$
 1: Method
 2: $\pi = \{F,\ Q - F\}$
 3: $S$ = the smallest of the sets $F$ and $Q - F$
 4: $\mathscr{L} = \{\}$
 5: for $a \in \Sigma$ do $\mathscr{L} = Append(\mathscr{L},\ (S, a))$
    end for
 6: while $\mathscr{L} \neq \{\}$ do
 7:   Extract $(S, a)$ in $\mathscr{L}$
 8:   Delete $(S, a)$ from $\mathscr{L}$
 9:   for $B \in \pi$ such that $B$ is refined by $(S, a)$ do
10:     $(B',\ B'') = (S, a)|B$
11:     Substitute in $\pi$ the block $B$ for $B'$ and $B''$
12:     $C$ = the smallest of the sets $B'$ and $B''$
13:     for all $b \in \Sigma$ do
14:       if $(B, b) \in \mathscr{L}$ then
15:         Update $\mathscr{L}$ by substituting $(B, b)$ for $(B', b)$ and $(B'', b)$
16:       else
17:         $\mathscr{L} = Append(\mathscr{L}, (C, b))$
18:       end if
19:     end for
20:   end for
21: end while
22: Return $(A/\pi)$
23: End Method.
```

oretical worst time-complexity of $\mathcal{O}(kn^2)$ for practical values of n. Nevertheless, the experimentation reported by Almeida et al. certainly shows that the time complexity of their incremental algorithm (when minimizing deterministic automata from an uniform distribution) does not depend on k. As mentioned above, this fact would make the algorithm competitive with respect to Hopcroft in some cases. The reason of this behavior is due to the depth-first traversal of the transition function and it makes the algorithm interesting in some practical circumstances, i.e. whenever the size of the alphabet is, at least, comparable to the number of states of the automaton [13]. Example 1 depicts the behavior of the algorithm.

Example 1 *Let us consider the automaton in Figure 1.*

The incremental algorithm initially stores the inequivalence of every pair in $(Q - F) \times F$. For this example, see Table 1. Then, the algorithm analyzes the equivalence of each pair of states for which it has not yet been

Algorithm 2.2 Incremental *DFA* minimization algorithm by Almeida et al.

```
Input: A DFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: The minimal DFA equivalent to $A$
 1: Method
 2: Let $M$ be a matrix indexed by $Q \times Q$ and initialize it to $undefined$
 3: for $(p, q) \in (Q - F) \times F$ do
 4:    $(p, q) = Normalize(p, q)$        // normalization of the pair of states
 5:    $M[p, q] = inequivalent$
 6: end for
 7: for $p \in Q$ do $M[p, p] = equivalent$
    end for
 8: Let $q_0, q_1, \ldots, q_n$ be an enumeration of $Q$
 9: for all $q_i \in Q$ $(0 \leq i \leq n - 1)$ do
10:    for all $q_j \in Q$ $(i + 1 \leq j \leq n)$ do
11:       $(q'_i, q'_j) = Normalize(q_i, q_j)$
12:       if $M[q'_i, q'_j] \neq undefined$ then Continue
          end if
13:       Initialize $EQUIV$ and $PATH$ to the empty list
14:       if $EquivP(q'_i, q'_j, EQUIV, PATH)$ then
15:          for $(p, q)$ in $EQUIV$ do $M[p, q] = equivalent$
             end for
16:       else
17:          for $(p, q)$ in $PATH$ do $M[p, q] = inequivalent$
             end for
18:       end if
19:    end for
20: end for
21: Return $A_{min}$ according the equivalences and inequivalences found.
22: End Method.
```

stablished. For each one of these pairs of states the algorithm recursively calls the equivalence test using as input the pairs of states reached with each one of the symbols in the alphabet.

Then, the algorithm considers the pair $(1, 5)$ *which leads to the analysis of pair* $(2, 7)$ *(note that* $\delta(1, a) = 2$ *and that* $\delta(5, a) = 7$*). The sequence of pairs traversed is shown in Figure 2. Taking into account that the pairs in the form* (p, p) *are equivalent, the sequence of pairs of states traversed can be summarized as shown below:*

$$(1, 5) \xrightarrow{a} (2, 7) \xrightarrow{b} (5, 9) \xrightarrow{b} (3, 10)$$

The fact that states 3 *and* 10 *are inequivalent, implies that all the pairs in the path from* $(1, 5)$ *to* $(3, 10)$ *are also inequivalent, and the algorithm marks them as such.*

Algorithm 2.3 *EquivP* function

```
Input: A pair of states $(p, q)$
Input: A list $PATH$ with already visited pairs of states
Input: A list $EQUIV$ with pairs of states detected as equivalent
Output: $True$ if both states are equivalent
    ($EQUIV$ would contain the list of new equivalent states)
Output: $False$ if both states are not equivalent
    ($PATH$ would contain the list of new inequivalent states)
 1: Method
 2: if $M[p, q] = inequivalent$ then return False
    end if
 3: if $(p, q) \in PATH$ then return True
    end if
 4: Append the pair $(p, q)$ to $PATH$
 5: for all $a \in \Sigma$ do
 6:   $(p', q') = Normalize(\delta(p, a), \delta(q, a))$
 7:   if $M[p', q'] = undefined$ and $(p', q') \notin EQUIV$ then
 8:     Append the pair $(p', q')$ to $EQUIV$
 9:     if not $EquivP(p', q', EQUIV, PATH)$ then
10:       return False
11:     else
12:       Remove the pair $(p', q')$ from $PATH$
13:     end if
14:   end if
15: end for
16: Append the pair $(p, q)$ to $EQUIV$
17: return True
18: End Method.
```

The algorithm considers then the pair $(1, 8)$ *which is found to be inequivalent because the pair* $(2, 8)$ *is already marked as such. The same happens with pairs* $(1, 9)$ *and* $(1, 10)$*. Table 2 depicts the current stage of the minimization.*

When the algorithm ends, the minimal DFA *for the language is output, that corresponds to the following partition of states:*

$$\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8, 10\}, \{9\}\}$$

Note that the algorithm also stores the equivalences found. This allows to partially minimize the input automaton without the need to finish the minimization process.

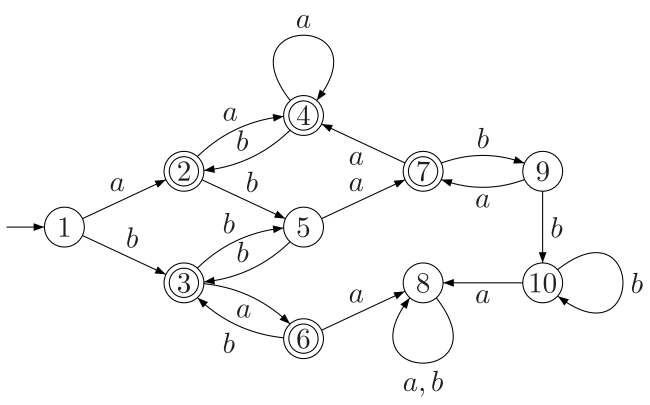

Figure 1: Automaton example.

	1	2	3	4	5	6	7	8	9	10
1	✓	X	X	X		X	X			
2		✓			X			X	X	X
3			✓		X			X	X	X
4				✓	X			X	X	X
5					✓	X	X			
6						✓		X	X	X
7							✓	X	X	X
8								✓		
9									✓	
10										✓

Table 1: Initial distinguishable (inequivalent) and equivalent pairs of states.

3 A hybrid algorithm

We present a new *DFA* minimization method that intends to improve the behavior of the algorithm by Almeida et al. summarized above. The minimization method shares features from both Hopcroft's and Almeida algorithm. It is described in Algorithm 3.1.

In the same way the incremental algorithm by Almeida et al. does, the algorithm analyzes each pair of states whose equivalence has not been stablished. The main difference with respect Almeida's algorithm is the following: whenever a pair of inequivalent states is found, instead of marking the pairs of states in the recursion stack (note that they are the pairs stored in $PATH$) as inequivalent, our algorithm successively refines the current partition π using a splitter (in the same way Hopcroft's algorithm does).

In order to store the evidence of the equivalences and inequivalences found, the algorithm keeps updated two partitions. Partition π is initialized

$$\begin{tikzcd}
 & & & \underset{6}{(3,10)} \\
\underset{1}{(1,5)} \arrow[dr, "a"] & & \underset{4}{(5,9)} \arrow[ur, "b"] \arrow[dr, "a"] & \\
 & \underset{2}{(2,7)} \arrow[ur, "b"] \arrow[dr, "a"] & & \underset{5}{(7,7)} \\
 & & \underset{3}{(4,4)} &
\end{tikzcd}$$

Figure 2: Pairs of states traversed in the automaton from pair $(1, 5)$. Small numbers show the order in which they are considered.

	1	2	3	4	5	6	7	8	9	10
1	✓	X	X	X	X	X	X	X	X	X
2		✓			X		X	X	X	X
3			✓		X			X	X	X
4				✓	X			X	X	X
5					✓	X	X		X	
6						✓		X	X	X
7							✓	X	X	X
8								✓		
9									✓	
10										✓

Table 2: Relationships found after the analysis of the pairs $(1, 5)$, $(1, 8)$, $(1, 9)$ and $(1, 10)$.

using an initial division of the states of the automaton into final and non-final states. Partition ρ contains the states which there is equivalence evidence for, and therefore it is initializes with as many singletons as states in the input *DFA*. These two partitions and a traversal of one of the two halves of the pairs of states matrix, make unnecessary the lists used by Almeida's et al. in their algorithm. Note that the normalization of the pairs of states are also unnecessary and that a plain ordering is enough to avoid extra computation (line 7 in Algorithm 3.2).

Thus, for each pair of states p and q whose equivalence has not been decided (states such that $[p]_\pi = [q]_\pi$ and $[p]_\rho \neq [q]_\rho$), the algorithm calls Algorithm 3.2 to carry out a depth-first analysis of the descendants of pair the pair (p, q). If a non-equivalence is found, then the algorithm returns a pair of non-equivalent states. The algorithm returns *True* if no such evidence is found.

Let us note that this algorithm uses a list $STATES$ to store the pairs of states reached while the analysis is in progress. The algorithm also stores

Algorithm 3.1 Hybrid algorithm

Input: A *DFA* $A = (Q, \Sigma, \delta, q_0, F)$

Output: The minimal *DFA* equivalent to A

```
1: Method
2: $\pi = \{Q - F, F\}$                                   // Non-equivalent states
3: $\rho = \{\{p\} \ : \ p \in Q\}$                          // Equivalent states
4: for all $(p, q) \in Q \times Q \ : \ [p]_\pi == [q]_\pi \wedge [p]_\rho \neq [q]_\rho$ do
5:   Initialize $STATES$ and $TODOLIST$ to empty lists
6:   Initialize $\Delta$ to an empty hashed list
7:   $out = AreEquivalent(p, q)$
8:   if $out \neq True$ then
9:     Append $out$ to $TODOLIST$
10:    while $TODOLIST \neq EMPTYLIST$ do
11:      Extract $(p', q')$, the first element in $TODOLIST$
12:      Delete $(p', q')$ from $STATES$
13:      for all $(p", q", a) \in \Delta_{(p', q')}$ do
14:        if $(p", q") \in STATES$ then
15:          if $[p"]_\pi == [q"]_\pi$ then
16:            Let $P$ = the smallest of the sets $[p']_\pi$ and $[q']_\pi$
17:            $\pi = \pi \wedge (P, a)|Q$
18:          end if
19:          Append $(p'', q'')$ to $TODOLIST$
20:        end if
21:      end for
22:    end while
23:  end if
24:  for all $(p', q') \in STATES$ do
25:    if $[p']_\rho \neq [q']_\rho$ then
26:      Update $\rho$ by merging the classes $[p']_\rho$ and $[q']_\rho$
27:    end if
28:  end for
29: end for
30: Obtain $A_{min}$ according the equivalences and inequivalences found.
31: Return $(A_{min})$
32: End Method.
```

Algorithm 3.2 *AreEquivalent* function

Input: A pair of states (p, q)

Output: A pair (p', q') of non-equivalent states

Output: $True$ when no evidence found of both states to be non-equivalent

```
 1: Method
 2: Append $(p, q)$ to $STATES$
 3: Initialize $EXPL$ to an empty stack
 4: for $a \in \Sigma$ do $Push(EXPL, (p, q, a))$
    end for
 5: while $EXPL \neq EMPTYSTACK$ do
 6:   $(p', q', a) = Pop(EXPL)$
 7:   Set $(p", q")$ to $Order(\delta(p', a), \delta(q', a))$
 8:   if $[p"]_\pi == [q"]_\pi$ then
 9:     if $[p"]_\rho \neq [q"]_\rho$ then
10:       if $(p", q") \notin STATES$ then
11:         Append $(p", q")$ to $STATES$
12:         for $b \in \Sigma$ do $Push(EXPL, (p", q", b))$
            end for
13:       end if
14:       if There is no element $(p', q', b)$ in $\Delta_{(p", q")}$ (where $b \in \Sigma$) then
15:         Add $(p', q', a)$ to $\Delta_{(p", q")}$
16:       end if
17:     end if
18:   else
19:     Append $(p", q")$ to $STATES$
20:     if There is no element $(p', q', b)$ in $\Delta_{(p", q")}$ (where $b \in \Sigma$) then
21:       Add $(p', q', a)$ to $\Delta_{(p", q")}$
22:     end if
23:     return $(p", q")$
24:   end if
25: end while
26: return $True$
27: End Method.
```

the ascendants of every visited pair (using a hashed list Δ). In this way, if $\Delta_{(q_1,q_2)}$ contains the tuple (p_1, p_2, a) then it means that from the pair (p_1, p_2) it is possible to reach the pair (q_1, q_2) using the symbol a (in other words, $Order(\delta(p_1, a), \delta(p_2, a)) = (q_1, q_2)$).

Essentially, the purpose of these data structures is to partially build a directed and acyclic graph which allows to maintain efficiently the information to, later, update the partitions π and ρ.

If Algorithm 3.2 concludes that the pair of input states is not equivalent it returns a pair (p, q) of non-equivalent states. Then, Algorithm 3.1 carries out a sequence of split operations that begins with the consideration of the pairs of states that reach (p, q). We note that this information is stored in $\Delta_{(p,q)}$ (Algorithm 3.2, line 10). Note that every split operation carried out may trigger new split operations. To avoid unnecessary splits to be carried out, the algorithm checks whether the operation is necessary or not (lines 14 and 15).

Once no more splits are possible, all the remaining pairs in the list $STATES$ denote two equivalent states. Algorithm 3.1 uses this information to accordingly update the partition ρ (line 24). Note that this implies the proposed algorithm to be incremental in the same way the incremental algorithm by Almeida et al. is.

Proposition 2 proves the correctness of the method and Example 3 depicts the behavior of the method. Proposition 4 proves the quadratic time complexity of the algorithm.

Proposition 2 *For any given* DFA A*, Algorithm 3.1 outputs the minimal* DFA *that accepts* $L(A)$.

Proof. *In order to prove the correctness of the method, first we take into account that the main loop considers only those states which no evidence has been found for (Algorithm 3.1, line 4). For each of those pairs, the subsequent call to the equivalence function decides its equivalence (or inequivalence). In any case, the algorithm ends when the equivalence is decided for the input pair of states.*

In order to avoid unnecessary loops, Algorithm 3.2 stores all visited pairs of states in the list $STATES$*. First, whenever Algorithm 3.2 finds a pair of inequivalent states* (p, q)*, the method has a splitter available to refine the partition* π *(there is a tuple* (p', q', a) *in* $\Delta_{(p,q)}$ *where both* $([p]_\pi, a)$ *and* $([q]_\pi, a)$ *are valid splitters able to distinguish the states* p' *and* q'*). Note that any splitting process may trigger new splits. Algorithm 3.1 updates the list* $TODOLIST$ *to maintain that information.*

Second, in the same way it is proved in [3], when Algorithm 3.2 returns $True$*, note that all the explored pairs of states are also equivalent, and therefore, partition* ρ *can be conveniently updated. Let us also note that, once a sequence of refinements of partition* π *has ended, all the remaining pairs in the* $STATES$ *list are equivalent, and therefore, the partition* ρ *can be also* *updated.*

Note also that this refinement/update process is repeatedly carried out taking into account every reached pair of states, and conclude when both partitions are equal. □

Example 3 *Let consider the automaton in Figure 3. Algorithm 3.1 uses partition ρ to store the evidence of equivalent states. Partition π is initialized with the trivially inequivalent states. The initial values of both partitions are:*

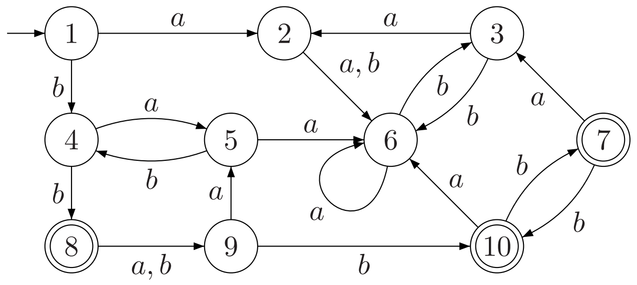

Figure 3: Example of automaton.

$$\pi = \{\{1,2,3,4,5,6,9\},\{7,8,10\}\}$$
$$\rho = \{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\{8\},\{9\},\{10\}\}$$

As we stated above, we note that the pairs of states (p',q') such that $[p']_\pi = [q']_\pi$ and $[p']_\rho \neq [q']_\rho$ are those for which there is no evidence and have to be analyzed.

Algorithm 3.1 first analyzes the pair $(1,2)$ and searches for a pair of inequivalent states[1]*. Figure 4 summarizes the sequence of pairs analyzed by the function AreEquivalent.*

Let us note that the method stores the information depicted in Figure 4 using the list STATES and the hashed structure Δ, whose values in this case are the following:

$$STATES = \{(1,2),(2,6),(3,6),(4,6),(5,6),(3,4),(2,5),(6,8)\}$$

$$\begin{array}{ll}
\Delta_{(1,2)}=\emptyset & \Delta_{(5,6)}=\{(4,6,a)\} \\
\Delta_{(2,6)}=\{(1,2,a),(3,6,a)\} & \Delta_{(3,4)}=\{(5,6,b)\} \\
\Delta_{(3,6)}=\{(2,6,b),(3,6,b)\} & \Delta_{(2,5)}=\{(3,4,a)\} \\
\Delta_{(4,6)}=\{(1,2,b),(2,5,b)\} & \Delta_{(6,8)}=\{(3,4,b)\}
\end{array}$$

Note that the states in the pair $(6,8)$ are not equivalent. The algorithm considers the tuples in $\Delta_{(6,8)}$ to construct the splitter. Thus, the smallest

[1]We note that, in order to obtain an easier-to-follow example of run, in line 12 of Algorithm 3.2 we stack the symbols of the alphabet in reverse order.

$$\begin{tikzcd}
 & (1,2) \arrow[r, "\substack{b\\(6)}"] \arrow[dd, "\substack{a\\(1)}"] & (4,6) \arrow[dd, "\substack{a\\(7)}"] & & \\
 & & & (3,4) \arrow[dr, "\substack{a\\(10)}"] \arrow[dd, "\substack{b\\(13)}"] & \\
 & (2,6) \arrow[ddl, bend left, "\substack{b\\(3)}"] \arrow[ddr, dashed, "\substack{a\\(2)}"'] & (5,6) \arrow[ur, "\substack{b\\(9)}"'] \arrow[dd, dashed, "\substack{a\\(8)}"] & & (2,5) \arrow[uull, bend right, "\substack{b\\(12)}"'] \arrow[ddll, dashed, bend left, "\substack{a\\(11)}"] \\
 & & & \underset{X}{(\mathbf{6},\mathbf{8})} & \\
(3,6) \arrow[uur, bend left, "\substack{a\\(4)}"] \arrow[loop left, "\substack{b\\(5)}"] & & (6,6) & &
\end{tikzcd}$$

Figure 4: Pairs of states traversed in the automaton from pair $(1, 2)$. Small parenthesized numbers note the order in which the analysis is carried out. Nodes reached by dashed edges are equivalent and are not considered. Bold pair (also marked with X) shows the first inequivalence found.

of the blocks $[6]_\pi = \{1, 2, 3, 4, 5, 6, 9\}$ *and* $[8]_\pi = \{7, 8, 10\}$ *together with the symbol* b *are used to split the current* π *partition. Using the splitter* $([8]_\pi, b)$, *and taking into account that* $\delta^{-1}(\{7, 8, 10\}, b) = \{4, 7, 9, 10\}$ *the result of the refinement of the partition is:*

$$\pi = \{\{1, 2, 3, 5, 6\}, \{4, 9\}, \{7, 10\}, \{8\}\}.$$

This process distinguish the states 3 *and* 4. *To keep triggering the refinement of the partition, the pair* $(3, 4)$ *is stored in* $TODOLIST$.

The method now consider the pair $(3, 4)$ *and uses the tuple* $(5, 6, b)$ *to obtain the splitter* $(\{4, 9\}, b)$ *because the block* $[4]_\pi = \{4, 9\}$ *is smaller than the block* $[3]_\pi = \{1, 2, 3, 5, 6\}$. *Thus, given that* $\delta^{-1}(\{4, 9\}, b) = \{1, 5, 8\}$, *the partition obtained is the following:*

$$\pi = \{\{1, 5\}, \{2, 3, 6\}, \{4, 9\}, \{7, 10\}, \{8\}\}.$$

The states 5 *and* 6 *are distinguished and the pair* $(5, 6)$ *is now stored in* $TODOLIST$.

The process continues considering the tuples in $\Delta_{(5,6)} = \{(4, 6, a)\}$. *Note that the states* 4 *and* 6 *have been already distinguished, therefore no split is necessary. The method detects it (Algorithm 3.1, line 15) and carries out no refinement. Nevertheless, the pair* $(4, 6)$ *is stored in* $TODOLIST$ *to continue the minimization process.*

Now the method analyzes the tuples in $\Delta_{(4,6)} = \{(1, 2, b), (2, 5, b)\}$. *Note that both the pairs* $(1, 2)$ *and* $(2, 5)$ *contain already distinguished pairs, Therefore, no split operation is carried out. Both pairs are stored in* $TODOLIST$.

The list $\Delta_{(1,2)}$ *is empty, therefore, the next pair of states considered is* $(2, 5)$ *and the tuple* $(3, 4, a) \in \Delta_{(2,5)}$. *Please note that the pair* $(3, 4)$ *has* *already been considered by the method. This fact is detected (Algorithm 3.1, line 14) and the splitting sequence of this iteration ends.*

The pairs that remain in the list $STATES$ contain equivalent states and the algorithm modifies the partition ρ accordingly. The partitions at the end of this iteration are shown.

$$\pi = \{\{1,5\},\{2,3,6\},\{4,9\},\{7,10\},\{8\}\}$$
$$\rho = \{\{1\},\{2,3,6\},\{4\},\{5\},\{7\},\{8\},\{9\},\{10\}\}$$

The method now considers the pair $(1,5)$. The analysis of this pair reaches the pair $(2,6)$ of equivalent states and the pair $(4,4)$. Therefore Algorithm 3.2 returns $True$ and the blocks $[1]_\rho$ and $[5]_\rho$ are merged. Now, the partitions are the following.

$$\pi = \{\{1,5\},\{2,3,6\},\{4,9\},\{7,10\},\{8\}\}$$
$$\rho = \{\{1,5\},\{2,3,6\},\{4\},\{7\},\{8\},\{9\},\{10\}\}$$

Please note that the next pair to be analyzed is $(4,9)$. The pairs $(5,5)$ and $(8,10)$ are reached. Note that the second pair contains two inequivalent states. Therefore, the smaller of the blocks $[8]_\pi = \{8\}$ and $[10]_\pi = \{7,10\}$ and the symbol b are considered to split the partition. Since $\delta^{-1}(\{8\},b) = \{4\}$, the current partitions are:

$$\pi = \{\{1,5\},\{2,3,6\},\{4\},\{7,10\},\{8\},\{9\}\}$$
$$\rho = \{\{1,5\},\{2,3,6\},\{4\},\{7\},\{8\},\{9\},\{10\}\}$$

The last pair to be analyzed is $(7,10)$. The process leads to detect that both states are equivalent, thus, the partition ρ is updated and the minimization of the automaton ends.

Proposition 4 *Algorithm 3.1 has $\mathcal{O}(kn^2)$ time complexity, where n and k denote the number of states and symbols of the automaton respectively.*

Proof. *In order to prove the complexity bound, please note that partitions π and ρ guide the calls to Algorithm 3.2 in the main loop of Algorithm 3.1 (line 7). In this way, Algorithm 3.2 will be called while these partitions were different.*

Let us stress that whenever, either the partition ρ or π (this by a split operation) are updated, it implies that, in the worst case, either partition π number of classes is increased by one, or partition ρ number of classes is decreased by one. Therefore, in the worst case, n of such operations are needed in order the partitions to be equal.

Let us now recall Proposition 5.3 in the paper by Berstel et al. [6], where the authors study Hopcroft's algorithm time-complexity. That proposition proves that, taking into account some data structures, the class to which a state belongs can be accessed in constant time, the enumeration of the elements of a class can be carried out in time proportional to the size of the *class, and also that the addition and deletion of an element in a class is done in constant time. Using these structures, the time complexity of splitting a partition is linearly-bounded with respect to the splitting set size. Other works also study the data structures to assure this (for instance [12, 14]).*

Let us mention that, using hash tables, both the insertion and search operations in the list $STATES$ can be carried out with constant time complexity. The implementation of Δ can consider double hashed tables in order to achieve constant access time. The remaining data structures can be implemented using stacks, also achieving constant time complexity of the operations.

We note that, for any given pair of states, Algorithm 3.2 explores the pairs of states accesibles with each symbol. Once a pair of states is visited, the relationship between the states in the pair will be taken into account in the partitions. Therefore, a pair of states will not be considered twice, and, no matter the number of calls, the total number of iterations of the loop in Algorithm 3.2 is bounded by $\mathcal{O}(kn^2)$.

Once a call to Algorithm 3.2 has ended, the pairs are used to either refine partition π or merge two blocks of partition ρ. We note that the method does not carry out unnecessary operations on partitions. We recall that, in order to minimize the input automaton, in the worst case, n of such operations are needed and that both operations have $\mathcal{O}(n)$ time complexity.

The number of refinement/merging operations carried out for each call to Algorithm 3.2 depends on the input automaton and cannot be apriori stablished. In the worst case, the equivalence function will have quadratic time complexity, but in that case, all the pairs of states are traversed, and only n of such pairs (in the worst case) are considered to modify the partitions. Therefore, the final complexity of the algorithm is $\mathcal{O}(kn^2)$.

$\square$

4 Experimental results

In order to test the behavior of the proposed algorithm, we used the same dataset [2] of uniformly distributed automata used in [3]. The dataset includes automata with 5, 10, 50 and 100 states and alphabet sizes of 2, 10, 25 and 50 symbols. For each configuration, 20.000 automata were considered. In [3], the authors state that this corpus ensures the results obtained to be statistically significant with a 99% confidence level within a 1% error margin.

All three algorithms were implemented in C++ (compiler MSVC 2013 Update 2 —Release x64 with LTCG on—). The experiments were carried out in an AMD Phenom X4 965 3.2 GHz with 4GB of RAM (DDR2

[2] According previous personal communication with the authors, the dataset was slightly modified in order to provide to each automaton in the corpus a uniformly-chosen set of final states.

800MHz), running a Windows 8.1 64bit operative system.

Table 3 shows the performance of the three algorithms studied: Hopcroft's; the incremental algorithm by Almeida et al. and our hybrid proposal. The performance is measured in terms of number of automata minimized per second.

$n = 5$				
	$k = 2$	$k = 10$	$k = 25$	$k = 50$
Hopcroft	338045	239171	158640	102432
Incremental	340501	338123	334446	329833
Hybrid	216983	207066	204630	199310

$n = 10$				
	$k = 2$	$k = 10$	$k = 25$	$k = 50$
Hopcroft	223237	139188	84125	50975
Incremental	209701	207680	202206	201479
Hybrid	123843	122177	119744	120365

$n = 50$				
	$k = 2$	$k = 10$	$k = 25$	$k = 50$
Hopcroft	52657	30127	17000	9826
Incremental	13238	12813	12789	12666
Hybrid	29786	30239	29777	29809

$n = 100$				
	$k = 2$	$k = 10$	$k = 25$	$k = 50$
Hopcroft	25018	13797	7689	4194
Incremental	1884	1798	1800	1793
Hybrid	14345	14948	15239	14982

Table 3: Experimental results (number of minimized DFAs per second).

We stress here the difference of the experimental results we show in this paper and the results reported in [3]. In the paper by Almeida et al. the incremental algorithm outperforms Hopcroft's algorithm regardless of the number of states or the size of the alphabet of the automata. This does not accord with the theoretical behavior of the algorithms. The higher the number of states of the automata the more obvious the divergence. The most feasible reasons for these discrepancies could be due to an unfair implementation of Hopcroft's algorithm.

Figure 5 summarizes the behavior of our implementation of the algorithms. The experimentation carried out with the dataset used in [3] shows that the performance of both the incremental and the hybrid algorithms does not depend on the size of the alphabet.

We would also like to stress that Hopcroft's algorithm always obtains the best results when the size of the alphabet is significantly smaller than the size of the automata. Figure 6 shows that, despite the initially worse performance of our proposal and the Hopcroft's algorithm, both methods improve the behavior of the algorithm by Almeida et al. from a given size of the automata on. This behavior is consistent with the theoretical asymptotic complexity of the algorithms. Furthermore, when the biggest alphabets of this dataset are taken into account, our hybrid proposal outperforms the method by Hopcroft, no matter the number of states of the automata (Figures 5 and 6).

Figure 7 shows that, regardless the size of the alphabet, the compared performance tendency between our proposal and the incremental algorithm increases, and therefore, it is expected that our algorithm to behave better for even bigger automata. Note also that the curves show both algorithms have the same dependence with respect to the size of the alphabet.

Let us recall that the main target of this study was the design of a split-minimization algorithm whose behavior would not be influenced by the size of the alphabet (as the algorithm by Almeida et al. is). This was the main reason to use the dataset in [3] to test the method.

For a given k value, it was expected that the hybrid algorithm would behave better than Hopcroft's method for automata with a number of states comparable to k. Nevertheless, for automata with a number of states significantly bigger than k, Hopcroft's algorithm would show better efficiency than our proposal.

Figure 8 represents the performance ratio between Hopcroft's and our algorithm. This figure shows that the number of states of the automata must be significantly bigger than the size of the alphabet in order Hopcroft's algorithm to be more efficient than our hybrid approach.

Let us here mention the interest of considering in an experimentation a set of uniformly distributed data (in our case automata). Nevertheless, in this case, it means that almost all the automata are already minimal, and it may cause some kind of bias in the experimentation.

5 Conclusions

In this work we present a new incremental algorithm to minimize deterministic finite automata that considers previous algorithms by Hopcroft and Almeida et al. Its design allows to take profit from intermediate results in order to reduce partially the input automaton, that could be interesting in some contexts.

The experimentation carried out in this paper took into account a set of automata selected using a uniform distribution of probability. When the method we propose is compared with the method by Almeida et al., the methods show that their running-time do not depend on the size of the alphabet. When bigger automata (in number of states) are considered, the experimentation shows that our method outperforms the approach by Almeida et al.

When our method is compared with Hopcroft's, and taking into account the ratio that relates the size of the alphabet with respect the number of states of the automata, the experimentation showed that the bigger the ratio the better compared behavior of the proposed algorithm.

References

[1] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company, 1979.

[2] B.W. Watson and J. Daciuk. An efficient incremental DFA minimization algorithm. *Natural Language Engineering*, 9(1):49–64, 2003.

[3] M. Almeida, N. Moreira, and R. Reis. Incremental DFA minimisation. In Michael Domaratzki and Kai Salomaa, editors, *CIAA*, volume 6482 of *Lecture Notes in Computer Science*, pages 39–48. Springer, 2010.

[4] J. E. Hopcroft. An $n \cdot \log n$ algorithm for minimizing states in a finite automaton. Technical report, Stanford University, Stanford, CA, USA, 1971.

[5] E. F. Moore. Gedanken experiments on sequential machines. In C. E. Shannon and J. Mc-Carthy, editors, *Automata Studies*. Princeton Universty Press, 1956.

[6] J. Berstel, L. Boasson, O. Carton, and I. Fagnot. *Automata: from Mathematics to Applications*, chapter Minimization of automata. European Mathematical Society. (arXiv:1010.5318v3). To appear.

[7] J. David. Average complexity of Moore's and Hopcroft's algorithms. *Theoretical Computer Science*, 417:50–65, 2012.

[8] M. Almeida, N. Moreira, and R. Reis. Aspects of enumeration and generation with a string automata representation. In Hing Leung and Giovanni Pighizzini, editors, *DCFS*, pages 58–69. New Mexico State University, Las Cruces, New Mexico, USA, 2006.

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

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

[11] N. Blum. A $\mathcal{O}(n \log n)$ implementation of the standard method for minimizing n-state finite automata. *Information Processing Letters*, 57:65–69, 1996.

[12] T. Knuutila. Re-describing an algorithm by Hopcroft. *Theoretical Computer Science*, 250:333–363, 2001.

[13] M. Veanes. Minimization of symbolic automata. Technical report, Microsoft Research, 2013. MSR-TR-2013-48.

[14] M. Lothaire. *Applied combinatorics on words*, chapter 1. Cambridge University Press, 2005.

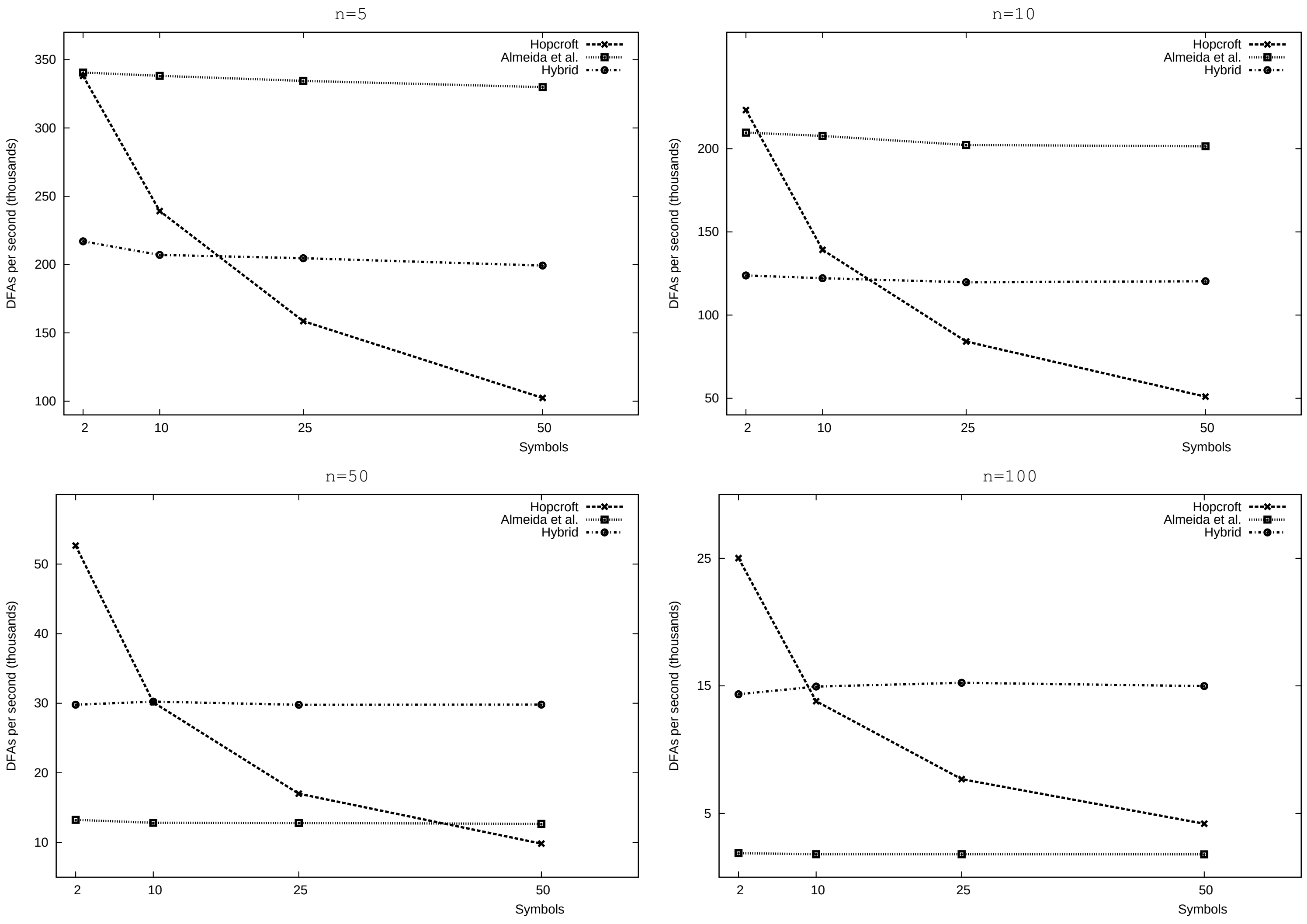

Figure 5: Performance of the algorithms (*DFA*s per second).

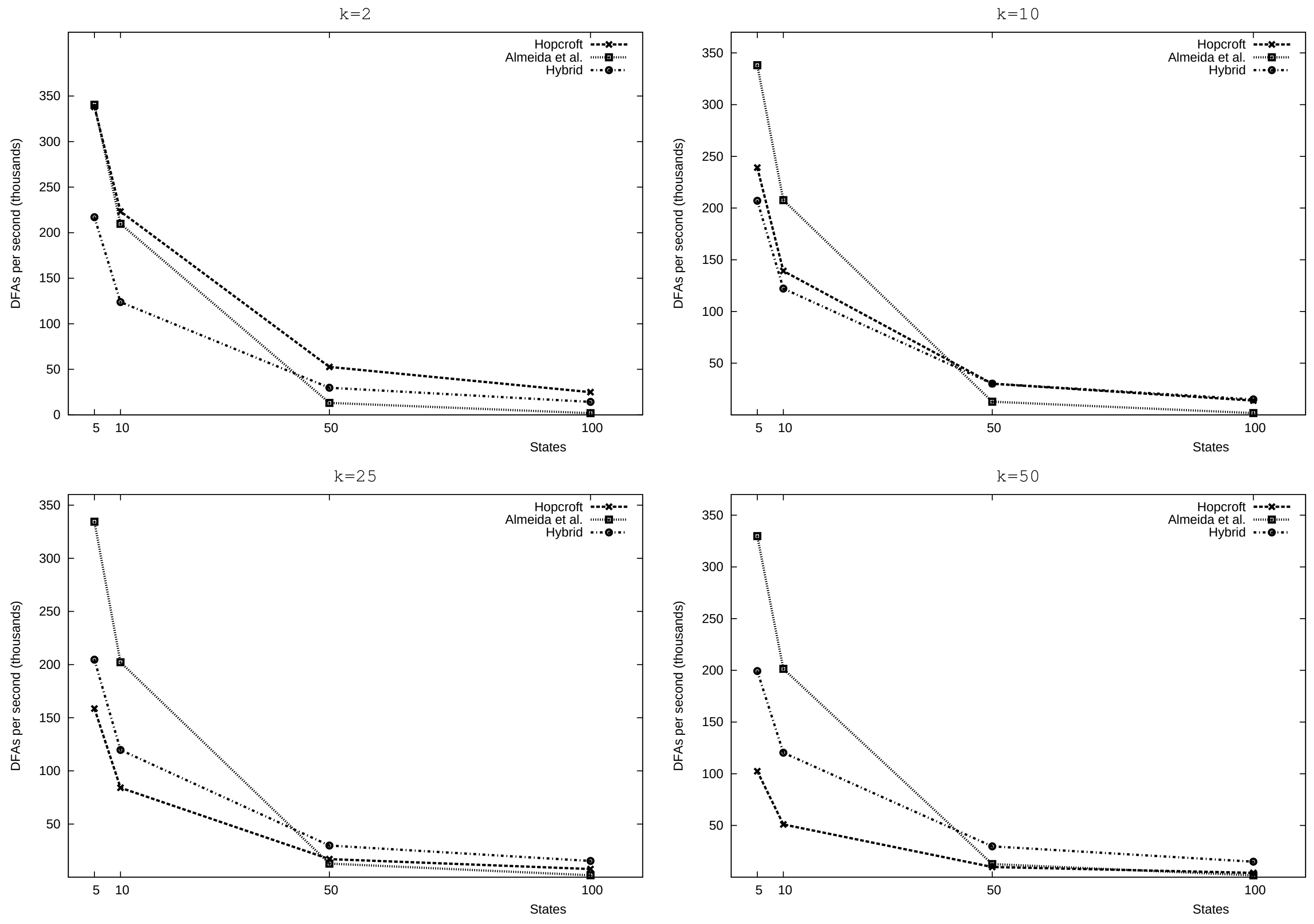

Figure 6: Performance of the algorithms (*DFA*s per second).

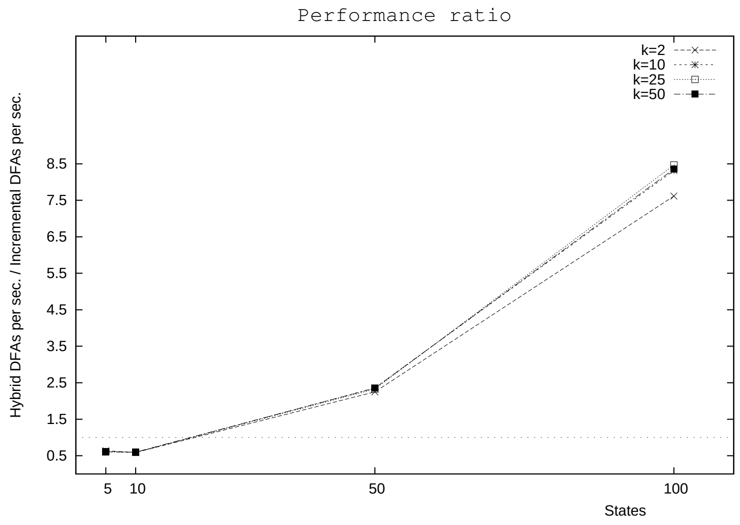

Figure 7: Compared performance between the incremental algorithm and our proposal (performance ratio).

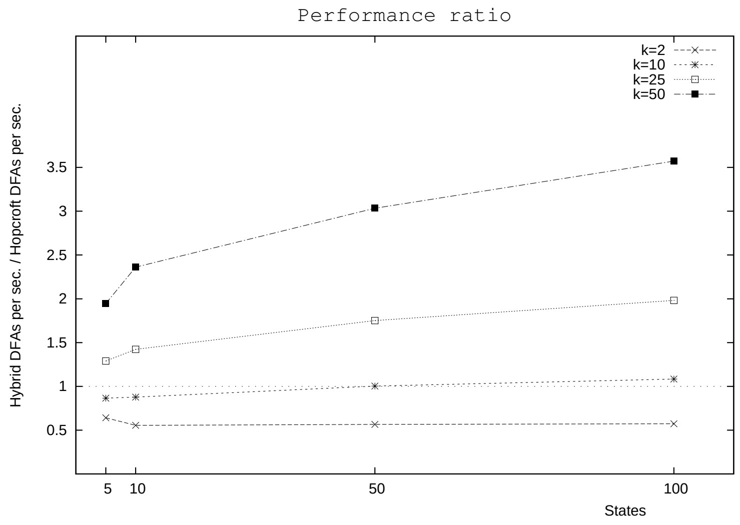

Figure 8: Compared performance between Hopcroft's algorithm and our proposal (performance ratio).