RAIRO-Theor. Inf. Appl. 48 (2014) 173–186
DOI: 10.1051/ita/2013045

Available online at:
www.rairo-ita.org

INCREMENTAL DFA MINIMISATION *

Marco Almeida[1], Nelma Moreira[1] and Rogério Reis[1]

Abstract. We present a new incremental algorithm for minimising deterministic finite automata. It runs in quadratic time for any practical application and may be halted at any point, returning a partially minimised automaton. Hence, the algorithm may be applied to a given automaton at the same time as it is processing a string for acceptance. We also include some experimental comparative results.

Mathematics Subject Classification. 68Q45, 68Q65, 68Q25.

1. Introduction

We present a new algorithm for incrementally minimising deterministic finite automata. This algorithm may be halted at any point, returning a partially minimised automaton that recognises the same language as the input. Should the minimisation process be interrupted, calling the incremental minimisation algorithm with the output of the halted process would resume the minimisation process. Moreover, the algorithm can be run on some automaton D at the same time as D is being used to process a string for acceptance. This can be useful, for example, on devices with limited resources such as embedded systems.

The algorithm uses a disjoint-set data structure to represent the DFA's states. Union-Find is used to mark pairs of equivalent states as well as to keep and update the equivalence classes. This approach maintains the transitive closure in a very concise and elegant manner. The pairs of states marked as distinguishable are stored in an auxiliary data structure in order to avoid repeated computations.

Keywords and phrases. Regular languages, finite automata, minimisation algorithms.

* *This work was partially funded by Fundação para a Ciência e Tecnologia (FCT), projects ASA, CANTE and PEst-C/MAT/UI0144/2011 through Programs COMPETE and FEDER.*

[1] Faculdade de Ciências, Universidade do Porto; {mfa,nam,rvr}@dcc.fc.up.pt

Unlike the usual approach to finite automata minimisation, which computes the equivalence classes of the set of states, this algorithm proceeds by testing the equivalence of pairs of states by continually following the transitions up to a point where it can be proven that all visited states are either equivalent or distinguishable. The intermediate results are stored for the speedup of ulterior computations, assuring quadratic running time and memory usage.

This paper is structured as follows. In the next Section some basic concepts and notation are introduced. Section 3 is a small survey of related work. In Section 4 we describe the new algorithm in detail, presenting the proofs of correctness and worst-case running-time complexity. Section 5 follows with experimental comparative results, and Section 6 finishes with some conclusions and future work.

2. Preliminaries

We recall here the basic definitions needed throughout the paper. For further details we refer the reader to the work of Hopcroft *et al.* [9].

An alphabet Σ is a nonempty set of symbols. A word over an alphabet Σ is a finite sequence of symbols of Σ. The empty word is denoted by ϵ and the length of a word w is denoted by $|w|$. The set $\Sigma^\star$ is the set of words over Σ. A language L is a subset of $\Sigma^\star$.

A *deterministic finite automaton* (DFA) is a tuple $D = (Q, \Sigma, \delta, q_0, F)$ where Q is finite set of states, Σ is the alphabet, $\delta : Q \times \Sigma \to Q$ the transition function, q_0 the initial state, and $F \subseteq Q$ the set of final states. We can extend the transition function to words $w \in \Sigma^\star$ such that $w = au$ by considering $\delta(q, \epsilon) = q$ and $\delta(q, w) = \delta(\delta(q, a), u)$, where $q \in Q$, $a \in \Sigma$, and $u \in \Sigma^\star$.

On any DFA, a state is called *accessible* if it is reachable from the initial state by some sequence of transitions. Similarly, all states that reach a final state are called *useful*. If all states of a DFA are accessible, we say it is *initially connected* (ICDFA).

The *language* accepted by the DFA D is $L(D) = \{w \in \Sigma^\star \mid \delta(q_0, w) \in F\}$. Two finite automata A and B are *equivalent*, denoted by $A \sim B$, if they accept the same language. For any DFA $D = (Q, \Sigma, \delta, q_0, F)$, let $\hat{\epsilon}(q) = 1$ if $q \in F$ and $\hat{\epsilon}(q) = 0$ otherwise, for $q \in Q$. Two states $q_1, q_2 \in Q$ are said to be *equivalent*, denoted by $q_1 \sim q_2$, if for every $w \in \Sigma^\star$, $\hat{\epsilon}(\delta(q_1, w)) = \hat{\epsilon}(\delta(q_2, w))$. Otherwise, $q_1, q_2 \in Q$ are said to be *distinguishable*. A DFA is *minimal* if there is no equivalent DFA with fewer states. Given a DFA D, the equivalent minimal DFA $D/_\sim$ is called the *quotient automaton* of D by the equivalence relation $\sim$. Minimal DFAs are unique up to isomorphism.

2.1. The Union-Find algorithm

The Union-Find [6, 12] algorithm takes a collection of n distinct elements grouped into several disjoint sets and performs two operations on it: merges two sets and finds to which set a given element belongs to. The algorithm is composed by the following three functions:

- MAKE(i) : creates a new set (singleton) for one element i (the identifier);
- FIND(i) : returns the identifier S_i of the set that contains i;
- UNION(i, j) : combines the sets identified by i and j into a new set formed by the union of those two sets; S_i and S_j are destroyed and removed from the collection.

An important detail of the UNION operation is that the two combined sets are destroyed in the end. Our implementation of the algorithm (using rooted trees) follows the one by Cormen *et al.* [6]. The main claim is that an arbitrary sequence of $m > n$ FIND instructions intermixed with $n - 1$ UNION instructions can be performed in $O(m\alpha(n))$ time, where $\alpha(n)$ is related to a functional inverse of the Ackermann function $A(x, y)$. And while $A(x, y)$ grows so fast that

$$A(4,3) = \overbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}^{65\,536 \text{ two's}},$$

$\alpha(n)$ is so slow-growing that, for all values of n such that $\log(n) < A(4, 3)$, $\alpha(n) \leq 3$. Thus, for all *practical* purposes, $\alpha(n) \leq 3$, and a sequence of $m > n$ FIND instructions intermixed with $n - 1$ UNION instructions can be performed in time $O(m)$.

3. Related work

The problem of writing efficient algorithms to find the minimal equivalent DFA can be traced back to the 1950's with the works of Huffman [10] and Moore [11]. Over the years several alternative algorithms have been proposed. In terms of worst-case complexity the best know algorithm (log-linear) is by Hopcroft [8]. Brzozowski [5] presented an elegant but exponential algorithm that may also be applied to non-deterministic finite automata.

To the best knowledge of the authors, the first DFA incremental minimisation algorithm was proposed by Watson [14]. The worst-case running-time complexity of this algorithm is exponential: $O(k^{\max(0,n-2)})$ for a DFA with n states over an alphabet of k symbols. As shown by Watson himself [13], this bound is tight. A later version of the algorithm, by Watson and Daciuk [15], employs a memoization technique to achieve an almost quadratic run-time. A bug was found, however, causing it to fail by returning non-minimal DFAs on some specific cases. One of the authors is currently working to fix it.

4. The incremental minimisation algorithm

Given an arbitrary DFA D as input, this algorithm may be halted at any time returning a partially minimised DFA that has no more states than D and recognises the same language. It uses a disjoint-set data structure to represent the DFA's states and the UNION-FIND algorithm to keep and update the equivalence classes. This approach allows us to maintain the transitive closure in a very concise and elegant manner. The pairs of states already marked as distinguishable are stored in an auxiliary data structure in order to avoid repeated computations.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA with $n = |Q|$ and $k = |\Sigma|$. We assume that the states are represented by integers, and thus it is possible to order them. This ordering is used to normalise pairs of states, as presented in Listing 1.

```
def NORMALISE-PAIR($p, q$):
    if $p < q$:
        $pair = (p, q)$
    else:
        $pair = (q, p)$
    return $pair$
```

LISTING 1. A simple normalisation step.

The normalisation step allows us to improve the behaviour of the minimisation algorithm by ensuring that only $\frac{n^2-n}{2}$ pairs of states are considered.

The quadratic time bound of the minimisation procedure DFA-MINIMISE-INCREMENTAL, presented in Listing 2, is achieved by testing each pair of states for equivalence exactly once. We assure this by storing the intermediate results of all calls to the pairwise equivalence-testing function INCREMENTAL-EQUIV-P, defined in Listing 3. Some auxiliary data structures, designed specifically to improve the worst-case running time, are presented on Section 4.1.

```
def DFA-MINIMISE-INCREMENTAL($D := (Q, \Sigma, \delta, q_0, F)$):
    for $q \in Q$:
        MAKE($q$)
    $\bar{E} := \{$NORMALISE-PAIR$(p, q) \mid p \in F, q \in Q - F\}$
    for $p \in Q$:
        for $q \in \{x \mid x \in Q, x > p\}$:
            if $(p, q) \in \bar{E}$:
                continue
            if FIND($p$) = FIND($q$):
                continue
            $E :=$ SET-MAKE($|Q|^2$)
            $H :=$ SET-MAKE($|Q|^2$)
            if INCREMENTAL-EQUIV-P($p, q$):
                for $(p', q') \in$ SET-ELEMENTS($E$):
                    UNION($p', q'$)
            else:
                for $(p', q') \in$ SET-ELEMENTS($H$):
                    $\bar{E} := \bar{E} \cup \{(p', q')\}$
    $C := \{\}$
    for $p \in Q$:
        $r :=$ FIND($p$)
        $C[r] := C[r] \cup \{p\}$
    $D' := D$
    JOIN-STATES($D', C$)
    return $D'$
```

LISTING 2. Incremental DFA minimisation in quadratic time. The SET operations are covered in Section 4.1

The algorithm Dfa-Minimise-Incremental starts by creating the initial equivalence classes (lines 2–3); these are singletons as no states are yet marked as equivalent. The global variable $\bar{E}$, used to store the distinguishable pairs of states, is also initialised (line 4) with the trivial identifications. Variables H and E, also global and reset before each call to Incremental-Equiv-P, maintain the history of calls to the transition function and the set of potentially equivalent pairs of states, respectively.

The main loop of Dfa-Minimise-Incremental (lines 5–18) iterates through all the normalised pairs of states and, for those not yet known to be either distinguishable or equivalent, calls the pairwise equivalence test Incremental-Equiv-P. Every call to Incremental-Equiv-P is conclusive and the result is stored either by merging the corresponding equivalence classes (lines 13–15), or updating $\bar{E}$ (lines 16–18). Thus, each recursive call to Incremental-Equiv-P will avoid one iteration on the main loop of Dfa-Minimise-Incremental by skipping (lines 7–10) that pair of states.

Finally, at lines 19–22, the set partition of the corresponding equivalence classes is created. Next, the DFA D is copied to D' (line 23) and the equivalent states are merged by the call to Join-States. The last instruction, at line 25, returns the minimal DFA D' equivalent to D.

```
 1  def Incremental-Equiv-P$(p, q)$:
 2      if $(p, q) \in \bar{E}$:
 3          return False
 4      if Set-Search$((p, q), H) \neq$ Nil:
 5          return True
 6      $H :=$ Set-Insert$((p, q), H)$
 7      for $a \in \Sigma$:
 8          $(p', q') :=$ Normalise-Pair(Find$(\delta(p, a))$, Find$(\delta(q, a))$)
 9          if $p' \neq q'$ and Set-Search$((p', q'), E) =$ Nil:
10              $E :=$ Set-Insert$((p', q'), E)$
11              if not Incremental-Equiv-P$(p', q')$:
12                  return False
13      $H :=$ Set-Remove$((p, q), H)$
14      $E :=$ Set-Insert$((p, q), E)$
15      return True
```

Listing 3. Pairwise equivalence test for Dfa-Minimise-Incremental.

Algorithm Incremental-Equiv-P is used to test the equivalence of the two states, p and q, passed as arguments.

The global variables E and H are updated with the pair (p, q) during each nested recursive call. As there is no recursion limit, Incremental-Equiv-P will only return when p is distinguishable from q (line 3) or when a cycle is found (line 5). If a call to Incremental-Equiv-P returns False, then all pairs of states recursively tested are distinguishable and variable H – used to store the sequence of calls to the transition function – will contain a set of distinguishable pairs of states. If it returns True, no pair of distinguishable states was found in the loop and variable E will contain a set of equivalent states. This is the strategy which assures that each pair of states is tested for equivalence exactly once: every call to Incremental-Equiv-P is conclusive and the result stored for future use. It does, however, lead to an increased usage of memory.

Theorem 4.1. *The algorithm* Dfa-Minimise-Incremental *is terminating.*

Proof. It suffices to notice the following facts:

- all the loops in Dfa-Minimise-Incremental are finite;
- the variable H on Incremental-Equiv-P assures that the number of recursive calls is finite. □

Theorem 4.2. *The algorithm* Incremental-Equiv-P *runs in* $O(kn^2)$ *time.*

Proof. The number of recursive calls to Incremental-Equiv-P is controlled by the global variable H. This variable keeps the history of calls to the transition function (line 6). In the worst case, all possible pairs of states are used: $\frac{n^2-n}{2}$, due to the normalisation step. Since each call may reach line 7, we need to consider k additional recursive calls for each pair of states, hence $O(kn^2)$. □

Lemma 4.3. *The algorithm* Incremental-Equiv-P *returns* True *if the two states passed as arguments are equivalent.*

Proof. Incremental-Equiv-P only returns True when $(p, q) \in H$ (lines 4–5) or a recursive call returned True (line 15). In both cases this means that a cycle with no distinguishable elements was detected, which implies that all the recursively visited pairs of states are equivalent. □

Lemma 4.4. *The algorithm* Incremental-Equiv-P *returns* False *if the two states passed as arguments are distinguishable.*

Proof. Given a pair of distinguishable states (p, q), all pairs of states (p', q') such that $\delta(p', w) = p$ and $\delta(q', w) = q$ are also distinguishable, for $w \in \Sigma^\star$. The global variable $\bar{E}$ contains all the pairs of states already proven to be distinguishable according to this criterion. Since Incremental-Equiv-P only returns False if the two states, p'' and q'' used as arguments, are such that $(p'', q'') \in \bar{E}$ (lines 2–3 and 12), the statement is correct. □

Theorem 4.5. *The algorithm* Incremental-Equiv-P *returns* True *if and only if the two states passed as arguments are equivalent.*

Proof. By direct application of Lemmas 4.3 and 4.4. □

Lemma 4.6. *At line 13 of* Dfa-Minimise-Incremental, *when* Incremental-Equiv-P *returns* True, *all the pairs of states stored in the global variable E are equivalent.*

Proof. By Lemma 4.3, if Incremental-Equiv-P returns True then the two states, p and q, used as arguments are equivalent. Since there is no depth recursion control, Incremental-Equiv-P only returns True when a cycle is detected. Thus being, all the pairs of states used as arguments in the recursive calls must also be equivalent. These pairs of states are stored in the global variable E at line 10 of Incremental-Equiv-P. □

Lemma 4.7. *At line 13 of* Dfa-Minimise-Incremental*, if* Incremental-Equiv-P *returns* False*, all the pairs of states stored in the global variable H are distinguishable.*

Proof. By Lemma 4.4, Incremental-Equiv-P returns False only when the two states, p and q, used as arguments are distinguishable. Throughout the successive recursive calls to Incremental-Equiv-P, the global variable H is used to store the history of calls to the transition function (line 6) and thus contains only pairs of states with a path to (p, q). All of these pairs of states are therefore distinguishable. □

Theorem 4.8. *Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$,* Dfa-Minimise-Incremental *computes the minimal DFA D' such that $D \sim D'$.*

Proof. The procedure Dfa-Minimise-Incremental finds pairs of equivalent states by exhaustive enumeration. The loop in lines 5–18 enumerates all possible pairs of states, and, for those not yet proven to be either distinguishable or equivalent, Incremental-Equiv-P is called. When line 19 is reached, all pairs of states have been enumerated and the equivalent ones have been found (cf. Thm. 4.5). The loop in lines 20–22 creates the equivalence classes and the procedure Join-States, at line 24, merges the equivalent states, updating the corresponding transitions. Since the new DFA D' does not have any equivalent states, it is minimal. □

Lemma 4.9. *Each time that* Incremental-Equiv-P *calls itself recursively, the two states used as arguments will not be considered in the main loop of* Dfa-Minimise-Incremental.

Proof. The arguments of every call of Incremental-Equiv-P are kept in two global variables: E and H.

By Lemma 4.6, whenever Incremental-Equiv-P returns True, all the pairs of states stored in E are equivalent. Immediately after being called from Dfa-Minimise-Incremental (line 13), if Incremental-Equiv-P returns True, the equivalence classes of all the pairs of states in E are merged (lines 14–15). Future references to any of these pairs will be skipped at lines 9–10.

In the same way, by Lemma 4.7, if Incremental-Equiv-P returns False, all the pairs of states stored in H are distinguishable. Lines 17–18 of Dfa-Minimise-Incremental update the global variable $\bar{E}$ with this new information and future references to any of these pairs of states will be skipped at lines 7–8 of Dfa-Minimise-Incremental. □

Claim 4.10. Algorithm DFA-MINIMISE-INCREMENTAL is incremental.

Proof. Halting the main loop of DFA-MINIMISE-INCREMENTAL at any point within the lines 5–18 only prevents the finding of *all* the equivalent pairs of states. Merging the known equivalent states on D', a copy of the input DFA D, assures that the size of D' is not greater than that of D and thus, is closer to the minimal equivalent DFA. Calling DFA-MINIMISE-INCREMENTAL with D' as the argument would resume the minimisation process, finding the remaining equivalent states. □

Theorem 4.11. *Algorithm* DFA-MINIMISE-INCREMENTAL *runs in* $O(kn^2\alpha(n))$ *time.*

Proof. The number of iterations of the main loop in lines 5–18 of DFA-MINIMISE-INCREMENTAL is bounded by $\frac{n^2-n}{2}$. Each iteration may call INCREMENTAL-EQUIV-P, which, by Theorem 4.2, is $O(kn^2)$. By Lemma 4.9 every recursive call to INCREMENTAL-EQUIV-P avoids one iteration on the main loop. Therefore, disregarding the UNION-FIND calls, and because all operations on variables $\bar{E}$, E, and H are $O(1)$, the $O(kn^2)$ bound holds. Since there are $O(kn^2)$ FIND and UNION intermixed calls, and exactly n MAKE calls, the time spent on all the UNION-FIND operations is bounded by $O(kn^2\alpha(n))$. All things considered, DFA-MINIMISE-INCREMENTAL runs in $O(kn^2 + kn^2\alpha(n)) = O(kn^2\alpha(n))$. □

```
def SET-MAKE($n$) :
    $T$ := HASH-TABLE($n$)
    $L$ := LINKED-LIST()
    return ($T$, $L$)

def SET-INSERT($v$, ($T$, $L$)) :
    $p$ := LIST-INSERT($v$, $L$)
    $T[v]$ := $p$
    return ($T$, $L$)

def SET-REMOVE($v$, ($T$, $L$)) :
    $p$ := $T[v]$
    LIST-REMOVE($p$, $L$)
    $T[v]$ := NIL
    return ($T$, $L$)

def SET-SEARCH($v$, ($T$, $L$)) :
    if $T[v] \neq$ NIL :
        $p$ := $T[v]$
        return LIST-ELEMENT($p$, $L$)
    else :
        return NIL

def SET-ELEMENTS(($T$, $L$)) :
    return $L$
```

Corollary 4.12. *Algorithm* DFA-MINIMISE-INCREMENTAL *runs in* $O(kn^2)$ *time for all* practical *values of* n.

Proof. Function α is related to an inverse of Ackermann's function. It grows so slowly that we may consider it a constant. □

4.1. Efficient set implementation

The variables E and H are heavily used in Incremental-Equiv-P as several insert, remove, and membership-test operations are executed throughout the algorithm. In order to achieve the desired quadratic upper bound, all these operations must be performed in $O(1)$. Therefore, in the following paragraphs, we describe some efficient set representation and manipulation procedures.

These set-manipulation procedures combine a hash-table with a doubly-linked list. This is another space-time trade-off that allows us to assure the desired complexity on all operations. The hash-table maps a given value (state of the DFA) to the address on which it is stored in the linked list. Since we know the size of the hash-table in advance – n^2 elements for a DFA with n states – searching, inserting, and removing elements is $O(1)$. The linked list assures that, at lines 14–15 and 17–18 of Dfa-Minimise-Incremental, the loop is repeated only on the elements that were actually used in the calls to Incremental-Equiv-P, instead of iterating through the entire hash-table.

The procedure Set-Make creates a new set represented by a tuple (T, L) where T is a hash-table and L a linked list. Its only argument is an integer defining the maximum size of the set. This information is necessary when the set is represented by a *direct-address table* ([6], pp. 222–223). If the number of states becomes too large for a single table, another option, somewhat more complicated, is *perfect hashing* ([6], pp. 245–249). Both representations assure $O(1)$ search, insert, and remove operations.

Set-Insert adds a new element to the set. The call to List-Insert (line 7) adds the element v to the linked list L, returning a pointer p to the memory address where v was stored. Next, at line 8, p is used as the value to the key v on the hash-table T. All these operations are performed in constant time. The updated set, represented by the tuple (T, L), is returned at the end of the procedure (line 9).

Removing an element v from a set is implemented by the procedure Set-Remove. Since the hash-table stores memory addresses, Set-Remove starts by obtaining, at line 12, the value p (memory address) of the key v. After using p to remove v from the linked list (line 13), p is replaced by the special value Nil as the pointer to the value v, meaning that v is no longer on the set. The updated set, without the element v, is returned at line 15.

The procedure Set-Search tries to locate an element v on a set (T, L). If the memory address of the element v (stored in the hash-table T) is valid, the pointer p is retrieved from T (line 19) and the corresponding value stored in the linked list is returned (line 20). If, on the other hand, there is no valid memory address for v (line 21), it is not an element of the set, and Nil is returned (line 22).

4.2. An example

Let us walk through the minimisation process of the DFA presented on Figure 1, exemplifying the incremental minimisation algorithm in action. It is a simple DFA

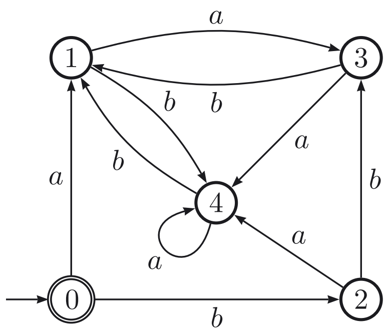

FIGURE 1. Incremental minimisation example: input DFA.

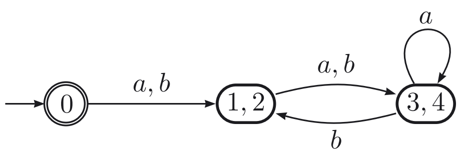

FIGURE 2. Incremental minimisation example: partially minimised DFA.

with 5 states, named sequentially from 0 to 4, over an alphabet of two symbols, a and b. The only final state (and also the initial state) is 0.

After creating the UNION-FIND data structures with lines 2–3, DFA-MINIMISE-INCREMENTAL will initialise the set of distinguishable pairs of states (variable $\bar{E}$ at line 4) with the trivial identifications: $\{ (0,1),(0,3),(0,2),(0,4) \}$.

Following the order induced by the states' names, the first pair to be tested for equivalence is $(0,1)$. These states are already known to be distinguishable and will be ignored at lines 7–8. The same is true for the pairs $(0,2)$, $(0,3)$, and $(0,4)$.

The next pair to be tested for equivalence is $(1,2)$. Nothing is known about this pair so INCREMENTAL-EQUIV-P is called. Following the transitions by a, the pair $(3,4)$ is reached. From $(3,4)$, regardless of the symbol used, a loop is reached: $(4,4)$ by a, and $(1,1)$ by b. Another loop is found when trying to follow the transitions of the states $(1,2)$ by the symbol b. This results on INCREMENTAL-EQUIV-P returning TRUE with $E = \{ (1,2),(3,4) \}$, and DFA-MINIMISE-INCREMENTAL making state 1 equivalent to state 2, and state 3 equivalent to state 4 by merging the corresponding UNION-FIND sets at lines 13–15. The partially minimised DFA obtained at this point is presented in Figure 2.

Continuing with the minimisation process, again, nothing is yet known about the equivalence of the states 1 and 3. The call to INCREMENTAL-EQUIV-P returns TRUE and sets the global variable $E = \{ (1,3),(2,4) \}$. After merging the equivalence classes of the states $(1,3)$ and $(2,4)$, the remaining pairs of states – $(1,4)$,

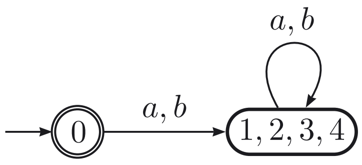

FIGURE 3. Incremental minimisation example: minimal DFA.

$(2,3)$, $(2,4)$, and $(3,4)$ – are known to be equivalent and are ignored at lines 9–10 of DFA-MINIMISE-INCREMENTAL.

Since there are no more states to test, line 19 of DFA-MINIMISE-INCREMENTAL is reached. The set partition corresponding to the equivalence classes is created (lines 20–22), and the equivalent states are merged (line 24). The resulting minimal DFA is presented in Figure 3.

5. EXPERIMENTAL RESULTS

In this Section we present some comparative experimental results on four DFA minimisation algorithms: Brzozowski, Hopcroft, Watson, and the new proposed incremental one. The results are presented in Tables 1–3.

Further details, including 3D graphics, complete tables with the exact values of the running time and memory usage for each algorithm may be found in Almeida [4].

All algorithms are implemented in the `Python` programming language and integrated in the FAdo project [7]. The tests were executed in the same computer, an Intel® Xeon® 5140 at 2.33 GHz with 4 GB of RAM, running a minimal 64 bit Linux system. We used samples of 20 000 automata, with $n \in \{5, 10, 50, 100\}$ states and alphabets with $k \in \{2, 10, 25, 50\}$ symbols. Since the data sets were obtained with a uniform random generator [1,2], the size of each sample is more than enough to ensure results statistically significant with a 99% confidence level within a 1% error margin. This is calculated with the formula $N = (\frac{z}{2\epsilon})^2$, where z is obtained from the normal distribution table such that $P(-z < Z < z)) = \gamma$, ϵ is the error margin, and γ is the desired confidence level.

Each test was given a time slot of 24 hours; processes that did not finish within this time limit were killed. Thus, and because we know how many ICDFAs were in fact minimised before each process was killed, the performance of the algorithms is measured in *minimised ICDFAs per second* (column Perf.). We also include a column for the memory usage (Space), which measures the peak value (worst-case) for the minimisation of the 20 000 ICDFAs in *kilobytes*.

The new incremental method always performs better. Both Brzozowski's and Watson's algorithm clearly show their exponential character, being in fact the only two algorithms that did not finish several minimisation tests. Hopcroft's algorithm,

TABLE 1. Experimental results for ICDFAs with $n \in \{5, 10\}$ states.

$n = 5$								
	$k = 2$		$k = 10$		$k = 25$		$k = 50$	
	Perf.	Space	Perf.	Space	Perf.	Space	Perf.	Space
Brzozowski	1458.47	4	142.57	4	54.56	4	27.80	4
Hopcroft	4810.67	4	1254.17	4	510.27	4	251.90	4
Watson	5754.74	4	2012.81	4	195.37	4	39.92	4
Incremental	6491.05	4	4557.84	4	2785.29	4	1830.87	4
$n = 10$								
	$k = 2$		$k = 10$		$k = 25$		$k = 50$	
	Perf.	Space	Perf.	Space	Perf.	Space	Perf.	Space
Brzozowski	92.11	4	2.55	4	0.00	3248	0.00	1912
Hopcroft	1378.91	4	274.55	4	112.89	4	54.11	4
Watson	1664.17	4	0.00	4	0.00	4	0.00	4
Incremental	3030.79	4	2362.92	4	1509.00	4	997.69	4

TABLE 2. Experimental results for ICDFAs with $n \in \{50, 100\}$ states.

$n = 50$								
	$k = 2$		$k = 10$		$k = 25$		$k = 50$	
	Perf.	Space	Perf.	Space	Perf.	Space	Perf.	Space
Brzozowski	0.00	664 704	0.00	799 992	0.00	1 456 160	0.00	750 312
Hopcroft	26.49	4	4.11	4	1.91	4	1.12	4
Watson	0.00	4	0.00	4	0.00	4	0.00	4
Incremental	225.83	4	173.18	4	131.11	4	85.88	4
$n = 100$								
	$k = 2$		$k = 10$		$k = 25$		$k = 50$	
	Perf.	Space	Perf.	Space	Perf.	Space	Perf.	Space
Brzozowski	0.00	2 276 980	0.00	1 061 556	0.00	961 144	0.00	2 862 312
Hopcroft	4.11	4	0.93	4	0.35	4	0.19	4
Watson	0.00	4	0.00	4	0.00	4	0.00	4
Incremental	32.45	4	27.63	4	23.66	4	19.60	4

TABLE 3. Experimental results for ICDFAs with 1000 states.

$n = 1000$						
	$k = 2$		$k = 3$		$k = 5$	
	Perf.	Space	Perf.	Space	Perf.	Space
Hopcroft	0.014	4	0.009	4	0.005	4
Incremental	0.47	74 196	0.42	74 180	0.43	74 220

although presenting a behaviour that appears to be very close to its worst-case, $O(kn\log(n))$, is always slower than the quadratic incremental method.

Due to the big difference between the measured performance of Hopcroft's algorithm and the new quadratic incremental one, we ran a new set of tests, only for these algorithms, using the largest ICDFA samples we have available. The results are presented in Table 3. These tests were performed on the exact same conditions as the previously described ones. Surprisingly, while Hopcroft's algorithm did not finish any of the batches within the time limit, it took only a little over 23 hours for the quadratic algorithm to minimise the sample of 20 000 ICDFAs with 1 000 states and 5 symbols. The memory usage of the incremental algorithm, however, is far higher: it always used nearly 74 MB, while Hopcroft's algorithm never required more than 4 kB.

6. Conclusions

We have presented a new incremental finite automata minimisation algorithm. Unlike other non-incremental minimisation algorithms, the intermediate results are usable and reduce the size of the input DFA. This property can be used to minimise a DFA when it is simultaneously processing a string or, for example, to reduce the size of a DFA when the running-time of the minimisation process must be restricted for some reason.

We believe that this new approach, while presenting a quadratic worst-case running-time, is quite simple and easy to understand and to implement. According to the experimental results, this minimisation algorithm outperforms Hopcroft's $O(kn\log(n))$ approach, at least in the average case, for reasonably sized automata.

Acknowledgements. We kindly thank Jan Daciuk for noticing and reporting to us an error on the implementation of Incremental-Equiv-P, found in a previous version of this paper that was presented at CIAA 2010 [3].

References

[1] A. Almeida, M. Almeida, J. Alves, N. Moreira and R. Reis, FAdo and GUItar: tools for automata manipulation and visualization, in vol. 5642 *14th CIAA'09*, edited by S. Maneth. *Lect. Notes Comput. Sci.* Springer (2009) 65–74.

[2] M. Almeida, N. Moreira and R. Reis, Enumeration and generation with a string automata representation, Special issue Selected papers of DCFS (2006). *Theoret. Comput. Sci.* **387** (2007) 93–102.

[3] M. Almeida, N. Moreira and R. Reis, Incremental DFA minimisation, in *Proc. of the 15th International Conference on Implementation and Application of Automata (CIAA 2010)* Winnipeg, MA, Canada, vol. 6482 of *Lect. Notes Comput. Sci.*, edited by M. Domaratzki and K. Salomaa. Springer-Verlag (2010) 39–48.

[4] M. Almeida, *Equivalence of regular languages: an algorithmic approach and complexity analysis*, Ph.D. thesis. University of Porto (2011).

[5] J.A. Brzozowski, Canonical regular expressions and minimal state graphs for definite events, in vol. 12 of *Proc. of the Sym. on Math. Theory of Automata*, edited by J. Fox. *MRI Symposia Series*, New York (1963) 529–561.

[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, *Introduction to Algorithms.* The MIT Press, 2nd edition (2003).

[7] Project FAdo, FAdo: tools for formal languages manipulation. `http://fado.dcc.fc.up.pt/, Access date:1.11.2011`.

[8] J. Hopcroft, An $n\log n$ algorithm for minimizing states in a finite automaton, in *Proc. Inter. Symp. on the Theory of Machines and Computations*, Haifa, Israel. Academic Press (1971) 189–196.

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

[10] D.A. Huffman, The synthesis of sequential switching circuits. *J. Symbolic Logic* **20** (1955) 69–70.

[11] E.F. Moore, Gedanken-experiments on sequential machines. *J. Symbolic Logic* **23** (1958) 60.

[12] R.E. Tarjan, Efficiency of a good but not linear set union algorithm. *J. ACM* **22** (1975) 215–225.

[13] B.W. Watson, *Taxonomies and toolkit of regular languages algortihms,* Ph.D. thesis. Eindhoven University of Tec. (1995).

[14] B.W. Watson, An incremental DFA minimization algorithm, in *International Workshop on Finite-State Methods in Natural Language Processing*. Helsinki, Finland (2001).

[15] B.W. Watson and J. Daciuk, An efficient DFA minimization algorithm. *Natur. Lang. Engrg.* (2003) 49–64.

Communicated by C. Choffrut.
Received August 11, 2012. Accepted December 11, 2013.