Natural Language Engineering **9** (1): 49–64. © 2003 Cambridge University Press
DOI: 10.1017/S1351324903003127 Printed in the United Kingdom

An efficient incremental DFA minimization algorithm

BRUCE W. WATSON

Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa
Software Construction Research Group, Technical University of Eindhoven, the Netherlands
Ribbit Software Systems Inc. & FST Labs
e-mail: watson@fst-labs.com

JAN DACIUK

Alfa-Informatica, Rijksuniversiteit Groningen, Groningen, the Netherlands
Technical University of Gdansk, Poland
e-mail: jandac@eti.pg.gda.pl

(*Received 27 March 2002; revised 1 November 2002*)

Abstract

In this paper, we present a new Deterministic Finite Automata (DFA) minimization algorithm. The algorithm is incremental – it may be halted at any time, yielding a partially-minimized automaton. All of the other (known) minimization algorithms have intermediate results which are not useable for partial minimization. Since the first algorithm is easily understood but inefficient, we consider three practical and effective optimizations. The first two optimizations do not affect the asymptotic worst-case running time – though they perform well on a large class of automata. The third optimization yields an quadratic-time algorithm which is competitive with the previously known ones.

1 Introduction

In this paper, we present a new Deterministic Finite Automata (DFA) minimization algorithm. The algorithm is incremental, meaning that it can be run on a DFA at the same time as the automaton is being used to process a string for acceptance. Furthermore, the minimization algorithm (hereafter called *the algorithm*) may be halted at any time, with the intermediate result being usable to partially minimize the DFA. All of the other (known) minimization algorithms have intermediate results which are not useable for partial minimization (Watson 1995).

It computes the equivalence of a given pair of states, drawing on techniques for determining structural equivalence of types in programming languages. The initial algorithm is easily understood and proven correct, though relatively inefficient. Later, optimization techniques, including memoization, are used to significantly improve the algorithm to running time quadratic in the number of states.

This paper is structured as follows: section 2 gives the mathematical preliminaries required for this paper; section 3 gives a characterization of minimality of a DFA;

section 4 gives a one-point algorithm which determines whether two states are equivalent; section 5 gives the incremental algorithm, making use of the one-point algorithm; section 6 introduces improvements to the original algorithm; section 7 presents the final algorithm including improvements; section 8 shows the impact of improvements on the performance of the algorithm; section 9 gives the closing comments for the paper. An early version of this algorithm was presented first in Watson (1995, 2001).

2 Mathematical preliminaries

We assume the reader has a solid grasp of formal language and automata theory – for such a background, see Hopcroft and Ullman (1979). A Deterministic Finite Automaton (DFA) is a 5-tuple $(Q, \Gamma, \delta, q_0, F)$, where Q is the finite set of states, Γ is the input alphabet, and $\delta \in Q \times \Gamma \longrightarrow Q \cup \{\perp\}$ is the transition function. It is actually a partial function, though we use $\perp$ to designate the invalid state. $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of *final* states. Throughout this paper, we consider a specific DFA $(Q, \Gamma, \delta, q_0, F)$.

The size of a DFA, $|(Q, \Gamma, \delta, q_0, F)|$, is defined as the number of states, $|Q|$.

To make some definitions simpler, we use the shorthand Γ_q to refer to the set of all alphabet symbols which appear as out-transition labels from state q. Formally, $\Gamma_q = \{a \mid a \in \Gamma \wedge \delta(q, a) \neq \perp\}$. We take $\delta^* \in Q \times \Gamma^* \longrightarrow Q \cup \{\perp\}$ to be the transitive closure of δ defined inductively (for state q) as $\delta(q, \varepsilon) = q$ and (for $a \in \Gamma_q$, $w \in \Gamma^*$) $\delta(q, aw) = \delta^*(\delta(q, a), w)$.

The right language of a state q, written $\overrightarrow{\mathscr{L}}(q)$, is the set of all words spelled out on paths from q to a final state. Formally, $\overrightarrow{\mathscr{L}}(q) = \{w \mid \delta^*(q, w) \in F\}$. With the inductive definition of δ^*, we can give an inductive definition of $\overrightarrow{\mathscr{L}}$:

$$\overrightarrow{\mathscr{L}}(q) = \left(\bigcup_{a \in \Gamma_q} \{a\} \overrightarrow{\mathscr{L}}(\delta(q, a)) \right) \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F \\ \emptyset & \text{if } q \notin F \end{cases}$$

Phrased differently, a word z is in $\overrightarrow{\mathscr{L}}(q)$ if and only if z is of the form az', where $a \in \Gamma$ is a label of an out-transition from q to $\delta(q, a)$ (i.e. $a \in \Gamma_q$), and z' is in the right language of $\delta(q, a)$, or $z = \varepsilon$ and q is a final state.

We define predicate *Equiv* to be "equivalence" of states: $Equiv(p, q) \equiv \overrightarrow{\mathscr{L}}(p) = \overrightarrow{\mathscr{L}}(q)$. With the inductive definition of $\overrightarrow{\mathscr{L}}$, we can begin rewriting *Equiv* as follows:

$$\begin{aligned}
& Equiv(p, q) \\
\equiv & \quad \langle \text{definition of } Equiv \rangle \\
& \overrightarrow{\mathscr{L}}(p) = \overrightarrow{\mathscr{L}}(q) \\
\equiv & \quad \langle \text{the inductive definition of } \overrightarrow{\mathscr{L}} \rangle \\
& (\varepsilon \in \overrightarrow{\mathscr{L}}(p) \equiv \varepsilon \in \overrightarrow{\mathscr{L}}(q)) \wedge \Gamma_p = \Gamma_q \wedge \\
& (\forall\, a : a \in \Gamma_p \cap \Gamma_q : \{a\} \overrightarrow{\mathscr{L}}(\delta(p, a)) = \{a\} \overrightarrow{\mathscr{L}}(\delta(q, a))) \\
\equiv & \quad \langle \text{the inductive definition of } \varepsilon \in \overrightarrow{\mathscr{L}}(p) \rangle
\end{aligned}$$

$$
\begin{aligned}
& (p \in F \equiv q \in F) \wedge \Gamma_p = \Gamma_q \wedge \\
& (\forall\ a : a \in \Gamma_p \cap \Gamma_q : \{a\}\overrightarrow{\mathscr{L}}(\delta(p,a)) = \{a\}\overrightarrow{\mathscr{L}}(\delta(q,a))) \\
\equiv \quad & \langle \text{for two languages } L_0, L_1 \colon (\{a\}L_0 = \{a\}L_1) \equiv (L_0 = L_1) \rangle \\
& (p \in F \equiv q \in F) \wedge \Gamma_p = \Gamma_q \wedge \\
& (\forall\ a : a \in \Gamma_p \cap \Gamma_q : \overrightarrow{\mathscr{L}}(\delta(p,a)) = \overrightarrow{\mathscr{L}}(\delta(q,a))) \\
\equiv \quad & \langle \text{definition of } \mathit{Equiv} \rangle \\
& (p \in F \equiv q \in F) \wedge \Gamma_p = \Gamma_q \wedge \\
& (\forall\ a : a \in \Gamma_p \cap \Gamma_q : \mathit{Equiv}(\delta(p,a), \delta(q,a)))
\end{aligned}
$$

We can extend the transition functions on pairs of states for $p, q \in Q$,$a \in \Sigma$,$w \in \Sigma^*$: $\delta(\{p,q\},a) \equiv \{\delta(p,a), \delta(q,a)\}$ and $\delta^*(\{p,q\},w) \equiv \{\delta^*(p,w), \delta^*(q,w)\}$.

3 Minimality of DFAs

The primary definition of minimality of a DFA M is:

$$(\forall\ M' : M' \text{ is equivalent to } M : |M| \leq |M'|)$$

where equivalence of DFAs means that they accept the same language. This definition of minimality is difficult to manipulate (in deriving an algorithm), and so we consider one written in terms of the right languages of states. Using right languages (and the Myhill-Nerode theorem – see section 3.4 of Hopcroft and Ullman (1979)), minimality can also be written as the following predicate:

$$(\forall\ p, q \in Q : p \neq q : \neg \mathit{Equiv}(p,q))$$

Additionally, we require that there are no useless states – those states which are not reachable from the start state and which cannot reach a final state; most DFA construction algorithms do not introduce useless states, so we ignore that issue in this paper. Armed with *Equiv*, we can determine whether two states are interchangeable, in which case one can be eliminated in favour of the other (of course, in-transitions to the eliminated state are redirected to the equivalent remaining one). We do not detail that reduction step in this paper, instead focusing on computing *Equiv*.

From the previous section, we have an inductive definition of *Equiv*. Since *Equiv* is an *equivalence relation* on states, we are actually interested in the greatest fixed point (in terms of refinement/containment of equivalence relations) of the equation $\mathit{Equiv}(p,q) \equiv$

$$(p \in F \equiv q \in F) \wedge \Gamma_p = \Gamma_q \wedge (\forall\ a : a \in \Gamma_p \cap \Gamma_q : \mathit{Equiv}(\delta(p,a), \delta(q,a)))$$

All of the known DFA minimization algorithms compute this fixed point from the *top side* (the *unsafe* side), meaning that until termination, they do not have a usable intermediate result (Watson 1995). The pointwise algorithm presented here computes it from below (the *safe* side).

4 A one-point algorithm computing $Equiv(p, q)$

Analogously to the problem of deciding the structural equivalence of two types, it is known that equivalence of two states can be computed recursively by turning the mutually recursive set of equivalences *Equiv* into a functional program. If the definition were to be used directly as a functional program, there is the possibility of non-termination in automata with cycles. For the functional program to work, it takes a third parameter along with the two states.

The program in Algorithm 4.1 computes relation *Equiv* pointwise[1]. Invocation $\mathsf{equiv}(p, q, \emptyset)$ returns $Equiv(p, q)$. During the recursion, it assumes that two states are equivalent (by placing the pair of states – unordered since *Equiv* is an equivalence relation – in S, the third parameter) until determined otherwise.

Algorithm 4.1:

```
func equiv(p, q, S) →
    if {p, q} ∈ S → eq := true
    [] {p, q} ∉ S →
       eq := (p ∈ F ≡ q ∈ F) ∧ (Γ_p = Γ_q);
       eq := eq ∧ (∀ a : a ∈ Γ_p ∩ Γ_q : equiv(δ(p, a), δ(q, a), S ∪ {{p, q}}))
    fi;
    return eq
cnuf
```

□

The ∀ quantification can be implemented using a repetition:

Algorithm 4.2:

```
func equiv(p, q, S) →
    if {p, q} ∈ S → eq := true
    [] {p, q} ∉ S → eq := (p ∈ F ≡ q ∈ F) ∧ (Γ_p = Γ_q);
                     if eq → for a : a ∈ Γ_p ∩ Γ_q →
                                 eq := eq ∧ equiv(δ(p, a), δ(q, a), S ∪ {{p, q}})
                             rof
                     [] ¬eq → skip
                     fi
    fi;
    return eq
cnuf
```

□

[1] It is similar to that presented in ten Eikelder (1991). The algorithm in that technical report computes structural equivalence of types in programming languages.

The correctness of the program in Algorithm 4.2 can be shown by extending the correctness argument given in ten Eikelder (1991). Naturally, the guard eq can be used in the repetition (to terminate the repetition when $eq \equiv false$) in a practical implementation. This optimization is omitted here for clarity.

There are a number of methods for making this program more efficient. From Watson (1995, section 7.3.3) and Wood (1987), it is known that the depth of recursion can be bounded by $(|Q| - 2)\,\mathbf{max}\,0$ without affecting the result. To track the recursion depth, we add a parameter k to function equiv such that an invocation $\mathsf{equiv}(p, q, \emptyset, (|Q| - 2)\,\mathbf{max}\,0)$ returns $Equiv(p, q)$. Purely for efficiency, the third parameter S is made a global variable; as a result equiv is no longer a functional program. The correctness of this transformation is shown in ten Eikelder (1991). We assume that S is initialized to $\emptyset$. When $S = \emptyset$, an invocation $\mathsf{equiv}(p, q, (|Q|-2)\,\mathbf{max}\,0)$ returns $Equiv(p, q)$; after such an invocation $S = \emptyset$.

Algorithm 4.3 (Pointwise computation of E):

```
func equiv(p, q, k) →
    if  k = 0 → eq := (p ∈ F ≡ q ∈ F)
    []  k ≠ 0 ∧ {p, q} ∈ S → eq := true
    []  k ≠ 0 ∧ {p, q} ∉ S →
        eq := (p ∈ F ≡ q ∈ F) ∧ (Γ_p = Γ_q);
        S := S ∪ {{p, q}};
        if  eq → for  a : a ∈ Γ_p ∩ Γ_q →
                        eq := eq ∧ equiv(δ(p, a), δ(q, a), k − 1)
                  rof
        []  ¬eq → skip
        fi;
        S := S \ {{p, q}}
    fi;
    return  eq
cnuf
```

□

动态规划程序的设计思想

To further improve the running time in practice, the procedure equiv can be *memoized*. Memoizing a functional program means that the parameters and the result of each invocation are tabulated in memory; if the function is invoked again with the same parameters, the tabulated return value is fetched and returned without recomputing the result. This resulting memoized algorithm was first given in Watson (2001).

5 The incremental algorithm

This latest version of function equiv can be used to compute $Equiv$. In variable Def_Ineq, we maintain the pairs of states known to be inequivalent (*distinguished*), while in Def_Equiv, we accumulate our computation of $Equiv$. To initialize Def_Ineq and Def_Equiv, we note that final states are never equivalent to nonfinal ones, and

that a state is always equivalent to itself. Since *Equiv* is an equivalence relation, we ensure that *Def_Equiv* is transitive at each step[2]. Finally, we have global variable S used in Algorithm 4.3):

Algorithm 5.1 (Computing *Equiv*):

$S, Def\_Ineq, Def\_Equiv := \emptyset, ((Q \setminus F) \times F) \cup (F \times (Q \setminus F)), \{(q,q) \mid q \in Q\};$
$\{\text{invariant: } Def\_Ineq \subseteq \neg Equiv \wedge Def\_Equiv \subseteq Equiv\}$
do $(Def\_Ineq \cup Def\_Equiv) \neq Q \times Q \rightarrow$
  let $p, q : (p,q) \in ((Q \times Q) \setminus (Def\_Ineq \cup Def\_Equiv))$;
  if $\mathsf{equiv}(p, q, (|Q| - 2)\,\mathbf{max}\,0) \rightarrow Def\_Equiv := Def\_Equiv \cup \{(p,q),(q,p)\};$
    $Def\_Equiv := Def\_Equiv^{+}$
  [] $\neg\mathsf{equiv}(p, q, (|Q| - 2)\,\mathbf{max}\,0) \rightarrow Def\_Ineq := Def\_Ineq \cup \{(p,q),(q,p)\}$
  fi
od; $\{Def\_Equiv = Equiv\}$
merge states according to *Def_Equiv*
$\{(Q, \Gamma, \delta, q_0, F)$ is minimal$\}$

□

The repetition in this algorithm can be interrupted and the partially computed *Def_Equiv* can be safely used to merge states.

Presently, it is the *only* known general incremental minimization algorithm.

5.1 Running time

Despite the fact that the depth of recursion in Algorithm 4.3 is $(|Q| - 2)\,\mathbf{max}\,0$, each invocation of function equiv potentially makes $|\Gamma|$ calls to itself. This gives a worst-case running time of $\mathcal{O}(\Gamma^{(|Q|-2)\,\mathbf{max}\,0})$ (exponential in the number of states) if we assume that $(p \in F \equiv q \in F) \wedge (\Gamma_p = \Gamma_q)$ and set updates (of S) can be done in constant time. This worst-case is difficult to achieve – though such an automaton is given in (Watson 1995). In the next section, we address this inefficiency and make some dramatic improvements.

6 Improvements

We describe three improvements to Algorithm 4.3 (hereafter called the "original" algorithm, since it was also given in Watson (2001)):

1. We pre-sort the states on their finality, the number of out-transitions, and labels on those transitions.
2. We put into S only pairs of states that can potentially start a cycle.
3. We introduce full memoization and demonstrate that the complexity of the algorithm is $\mathcal{O}(|Q|^2 G(|Q|^2))$, where $G(n)$ is the inverse of Ackermann's function. ($G(n) \leq 5$ for all "practical" values of n, i.e. for all $n \leq 2^{2^{16}}$ – see Aho, Hopcroft and Ullman (1974) for details.)

[2] Since *Def_Equiv* is initially the identity relation on states, it is already reflexive.

6.1 Pre-sorting

The original algorithm divides states into two classes: final and non-final states. Only pairs of states each belonging to the same class are tested for equivalence.

We propose to divide the set of states into more (numbered) classes[3]. The criterion includes not only the finality of states, but also the number of out-transitions, and their labels. This can be done in $\mathcal{O}(|Q|)$ time using bucket sort. bucket sort 桶排序

It brings several advantages:

- As we compare the states pairwise only in their classes (states from two different classes are not equivalent), there are fewer comparisons, e.g. if we divide the set into n roughly equally numerous classes, we perform n times fewer comparisons.
- Instead of comparing the number of out-transitions and their labels in line 5 of the algorithm, we compare only classes of states (i.e. two integer numbers)[4].
- We need less memory to represent non-equivalent states, as there is no need to remember that states belonging to different classes are not equivalent.

If all states in the automaton have the same number of transitions, and the transitions have the same labels (for example, in a complete automaton), there is nothing to gain with pre-sorting.

6.2 Remembering only re-entrant states on stack

In line 6 of the original algorithm, the currently examined pair of states is added to variable S (the stack of pairs of states – predecessors of the currently examined pair), and a few lines later, the pair is removed from S.

The purpose of variable S is to detect cycles. There are two ways in which those cycles can be started:

- a single chain of transitions may lead from the initial pair back to it – the cycle is started at the initial pair;
- the cycle starts later on, but it can only start from a state that has more than one in-transition; one in-transition leads from a path from a state in the initial pair (outside the cycle), so there must be at least another in-transition from within the cycle.

There is no need to add to the stack pairs of states that cannot start a cycle. It is guaranteed that we cannot revisit such a pair during the same call to equiv – we would have to revisit a pair that starts a cycle first (and then there would be no need to follow out-transitions, as the result would already be known).

The profit from the improvement is twofold: (1) we use less memory to represent S in memory efficient implementations; and (2) in some memory-efficient implementations, searching S takes $\mathcal{O}(\log|S|)$ time, contributing that factor to the overall

[3] Pre-sorting on finality and the number of transitions was also suggested by Lauri Karttunen.
[4] This is impossible if we sort only on finality and number of out-transitions.

complexity, so shortening S means also shortening the time. If (almost) all states of the automaton have more than one in-transition, this modification does not reduce the running time, nor memory requirements.

6.3 Full memoization

In the previous section, we can see that the worst-case computational complexity is $\mathcal{O}(\Gamma^{(|Q|-2)\max 0})$ even when partial memoization is used. When full memoization[5] is used, the complexity is actually $\mathcal{O}(|Q|^2 G(|Q|^2))$ – far better than exponential.

A call to function equiv can return only two results: *true* or *false*. However, we can immediately memoize only the non-equivalence. Storing and retrieving that information can be done in constant time in two-dimensional arrays indexed with states numbers.

If function equiv returns *true*, the result can either be conclusive, or inconclusive. If the result depends upon the equivalence of a certain pair of states that is still under evaluation, i.e. it is still in variable S, the result is inconclusive.

A call to function equiv with $\{p,q\}$ as an argument may create various situations:

- $\{p,q\}$ is in fact $\{p,p\}$ – equiv returns conclusive true (note that the pair may have originally been different, but the states could have been merged in a different branch of the same top-level call to equiv);
- $\{p,q\}$ are already remembered as non-equivalent, i.e. $\{p,q\} \in Def\_Ineq$ – equiv returns false;
- a descendant of $\{p,q\}$ is already under evaluation higher up in the call hierarchy ($\exists_{w\in\Sigma^*}\ \delta^*(\{p,q\},w) = \{r,s\}$ such that $\{r,s\} \in S\backslash\{p,q\}$) – equiv returns inconclusive true, making all pairs in the call hierarchy from $\{p,q\}$ up to and including $\{r,s\}$ dependent on $\{r,s\}$; if other pairs of states depend on $\{p,q\}$, those dependencies are transfered to $\{r,s\}$ using the UNION-FIND algorithm;
- $\{p,q\}$ is not in S, but it is on a list of dependent states for a pair of states $\{r,s\}$ in S – equiv returns inconclusive true, making all pairs up to and including $\{r,s\}$ depend on at least $\{r,s\}$; note that no additional dependencies have to be stored at this point;
- any of pairs of states immediately reachable from $\{p,q\}$ is inequivalent – $\{p,q\}$ is added to Def_Ineq, and equiv returns false. If there are any pairs depending on $\{p,q\}$, they are added to Def_Ineq.
- $\{p,q\}$ depends only upon $\{p,q\}$, i.e. $\exists_{w\in\Sigma^*}\ \delta^*(\{p,q\},w) = \{p,q\}$ and $\forall_{w\in\Sigma^*}\ \delta^*(\{p,q\},w) \notin (S\backslash\{p,q\})$ – all pairs depending on (and including) $\{p,q\}$ are merged, and equiv returns true.

Consider the automaton in figure 1. Function equiv has been called to determine the equivalence of states 2 and 6. It depends upon the equivalence of the pair of

[5] We memoize the result only with respect to the pair of states; we ignore the depth of recursion. However, we call it *full* memoization here, as the depth of recursion is irrelevant for evaluating equivalence of a pair of states.

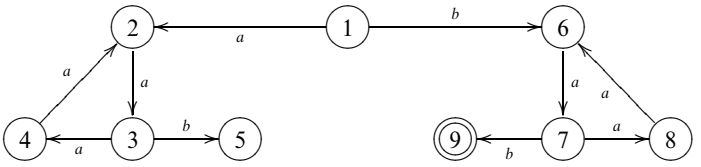

Fig. 1. Inconclusive comparison of states 4 and 8, resulting from evaluation of the pair 2 and 6. It is assumed that states 5 and possibly 9 have further outgoing transitions – otherwise states 2–5 could simply be deleted. 1 is the start state.

states $\{3,7\}$, so that pair is visited as well. Assuming that we traverse the transitions in lexicographical order, the algorithm goes on to examine the pair $\{4,8\}$. From $\{4,8\}$, it moves to $\{2,6\}$ to see that that pair is already in S, so the call to equiv$(2,6,3)$ with $S = \{\{2,6\},\{3,7\},\{4,8\}\}$ returns *true*. However, neither in $\{2,6\}$, nor in $\{4,8\}$ can we say that those pairs are equivalent. They are only potentially equivalent. The pair $\{2,6\}$ is still under investigation – we have not checked whether $\{3,7\}$ is equivalent (the call to equiv$(3,7,6)$ has not returned any value yet), and the pair $\{4,8\}$ depends upon that evaluation. In case of $\{2,6\}$ we could simply wait until the nested calls return – we will know the result when it has been computed. However, in case of $\{4,8\}$ we have no other call to equiv higher up.

We remember that a pair of states $\{p,q\}$ depends upon another pair $\{r,s\}$ higher up in the hierarchy of recursive calls to function equiv. When the control returns to the first call with the $\{r,s\}$ as the argument, there are two possibilities. Either the pair does not depend upon any other pair higher up in the hierarchy (it can depend on itself), or such dependence exists. In the first case, all pairs that depend on $\{r,s\}$ are either merged (if pair $\{r,s\}$ is equivalent), or remembered as non-equivalent (if pair $\{r,s\}$ is not equivalent). If the pair $\{r,s\}$ depends upon another pair, a new dependency is stored. To prevent formation of chains of dependencies, we use the UNION-FIND algorithm (Aho, Hopcroft and Ullman 1974). Dependencies for pairs are stored in two-dimensional arrays as pointers to structures containing the representative (the pair highest in the hierarchy), and a list of dependant pairs.

7 The final algorithm

Figure 2 shows the final version of the algorithm. Depending on memory limitations and speed requirements, certain operations can be implemented in different ways. We have provided two implementations: one making use of two-dimensional arrays (achieving desired complexity), and another using tree structures (with $\log|Q|$ overhead). It is possible to use hash tables in an implementation. This would combine the speed of the first solution with memory efficiency of the other one.

When memoization is used, function equiv can still be called more than once with the same pair of states. If it has already been called with the same pair, lines 2–7 check this, and the control never reaches line 8. In line 2, states are compared for identity. Although at the top level, equiv is never called with a pair of identical states, the states might be identical on other levels. The check is done in constant

```
(1)   func equiv(p, q, k)
(2)       if p = q then eq := true
(3)       elsif class[p] ≠ class[q] then eq := false
(4)       elsif k = 0 then eq := true; rl := 0
(5)       elsif {p, q} ∉ S then eq := true; rl := index({p, q}, S)
(6)       elsif {p, q} ∈ Def_Ineq then eq := false
(7)       elsif {p, q} ∈ P then eq := true; rl := index({p, q}, P)
(8)       else
(9)           if level = 0 ∨ in(p) > 1∨ in(q) > 1 then
(10)              S := S ∪ {{p, q}}; level := level +1; pushed := true
(11)          fi;
(12)          rl′ := |Q|; eq := true;
(13)          for a : a ∈ Γ_p ∩ Γ_q do
(14)              eq := eq ∧ equiv(δ(p, a), δ(q, a), k − 1);
(15)              rl′ := min(rl′, rl)
(16)          rof;
(17)          rl := rl′;
(18)          if pushed then
(19)              S := S \ {{p, q}}; level := level −1
(20)          fi;
(21)          if eq then
(22)              if rl > level then
(23)                  merge({p, q})
(24)              else
(25)                  P[rl] := P[rl] ∪ {{p, q}}
(26)              fi
(27)          else
(28)              def_Ineq := Def_Ineq ∪ {{p, q}}
(29)          fi;
(30)          if rl = level then rl := |Q| fi;
(31)          if eq then
(32)              if rl = |Q| then
(33)                  ∀{r,s}∈P[level] merge({r, s})
(34)              else
(35)                  P[rl] := P[rl] ∪ P[level]
(36)              fi
(37)          else
(38)              ∀{r,s}∈P[level] Def_Ineq := Def_Ineq ∪ {{r, s}}
(39)          fi;
(40)          P[level] := ∅
(41)      fi;
(42)      return eq
(43)  cnuf
(44)
```

Fig. 2. Comparison of a pair of states in the incremental minimization algorithm – final version.

time. Line 3 tests whether the states are either both final, or both non-final, and whether they have the same number of transitions, and the same labels on those transitions. It is equivalent to line 5 of the original algorithm. However, because of pre-sorting, there is no need to compare so many features each time; only class numbers need to be compared. The check is done in constant time.

Line 4 is the same as in the original algorithm, except for variable *rl* that is set to 0. The variable indicates the level of recursion at which a pair of states can be found that the current pair depends upon. If S is seen as stack, then variable *rl* is an index in that stack, indicating a position of the pair the current pair depends upon. If the current pair depends on nothing, variable *rl* is set to $|Q|$. If the comparison has done a full cycle, it means that the current pair of states depends on the initial pair of states (the one from the top level call). This check is done in constant time. Line 5 belongs to the same category – it also tests for cycles. If a cycle has been detected, the level of the pair on which the current pair depends is stored in global variable *rl*. That variable propagates the dependency upwards. This check can be done in constant time.

In line 6, it is checked whether the pair is already known not to be equivalent. Variable *Def_Ineq* is adopted for that purpose. The check can be done in constant time. In line 7, it is checked whether the pair has already been found to depend upon another pair higher up in the hierarchy. While searching for the pair, the dependency chains are updated (see section 6.3). This can be done in $\mathcal{O}(G(|Q|))$ time ($|Q|$ is the maximal depth of recursion, so also the maximal height of dependency tree). This concludes memoization checking.

Lines 9–11 update the value of global variable S. Only the pair of states from the top level calls, and pairs where at least one state has more than one in-transition, need to be put into S. The update can be done in constant time. The pair is removed in lines 18–20, which can also be done in constant time.

In lines 12–17, it is checked whether the states are equivalent by checking the equivalence of the targets of their out-transitions. This is the central part of the original algorithm. As variable *rl* is global, so it is used to convey any dependency upwards in the call hierarchy. Variable rl' is local, so it can store the local value between the nested calls.

If the current pair of states has been found to be equivalent, then the result can be conclusive or not. If the result is positive and conclusive, i.e. the equivalence of $\{p, q\}$ does not depend upon equivalence of any other pair of states, then the states are merged (line 23). It is done using the UNION-FIND algorithm with complexity $O(G(|Q|))$. If the result is positive, but not conclusive, then the result must be stored along with the pair it depends upon (line 25). It is appended to the list for appropriate level, and a mapping from $\{p, q\}$ to that other pair is stored. If the pair is not equivalent, it is added to the set of non-equivalent pairs *Def_Ineq* in line 28. If *Def_Ineq* is a two-dimensional array, this can be done in constant time.

There can be pairs of states that depend upon the current pair of states $\{p, q\}$. They are stored in $P[level]$, where *level* is the recursion level (or the number of items in S) for the current pair. If $\{p, q\}$ are found to be equivalent, then if the result is conclusive, the pairs are merged. There can be more than one pair of states in $P[level]$, but the operation cannot be invoked more than $|Q|^2$ times across all calls. The operation can be performed in time proportional to the number of items in $P[level]$. If the result is not conclusive, the pairs are merged with the pairs at $P[rl]$. Appending a list to another list (moving it to the end of another list – without

copying) can be done in constant time. If $\{p, q\}$ are found not to be equivalent, the contents of $P[level]$ is stored in Def_Ineq – again, it is possible to do that in constant time for each pair. There can be at most $|Q| \cdot (|Q| - 1)/2$ pairs across all calls, so the cost per pair, or per average call, is constant.

Function equiv is called at most $|Q|^2 \, \mathbf{max}_{q \in Q} |\Gamma_q|$ times. One call (without nested calls) can be completed in constant time, except for certain operations on lists stored in P. However, all of these operations take at most $\mathcal{O}(|Q|^2)$ time across all calls, except for the UNION-FIND algorithm that applied to both dependencies and equivalent states takes $|Q|^2 G(|Q|^2)$ across all calls, so the final complexity is $\mathcal{O}(|Q|^2 G(|Q|^2))$.

8 Results

The performance was checked for various implementations of the algorithm. Not only the speed of execution was measured, but also some additional characteristics, like the maximal depth of recursion of function equiv, the maximal number of items in variable S, and the number of calls to function equiv excluding memoization. They were measured for several versions of the algorithm: the standard (final) one, one where two-dimensional arrays were replaced with tree structures, one without pre-sorting, and one without full memoization (i.e. the original version).

8.1 Input data

Experiments were performed on four different data sets. The first one contained determinized versions of automata used in van Noord (2000). They originally come from experiments on finite-state approximation of context-free grammars – a real life task. The automata have varied characteristics. However, they are of moderate size, so only the biggest ones are suitable for measuring the effects of various improvements on the speed of the algorithm. Examining the Table 2, we can see that a version of the algorithm that uses tree structures is usually faster than the one using two-dimensional arrays. A quick look at Table 1 solves the mystery. The automata have relatively many classes, so the number of items in variable S is low, and initialization for two-dimensional arrays takes more time than the use of tree structures.

The other three sets were generated automatically. The first generated set consisted of large automata with a large number of classes (in the sense defined in the section 6.1). This means that most of the job of minimalization was done in the sorting phase. Also for these automata, tree structures were faster than two-dimensional arrays.

The second generated data set consisted of large automata with a fixed low number of classes. However, the automata were already minimal. For this kind of job, most of the work was done by the function equiv.

The third generated data set consisted of automata from the second generated set inflated to a larger size. A program was used to create additional states in an automaton without changing its language. The automata still had the same, low

Table 1. *Results of experiments on real examples.* $|Q|$ *– number of states,* $|\delta|$ *– number of transitions, cl – number of classes, min – number of states in minimal automaton,* S_o *– max depth of recursion of* equiv, $S_n - S_o$ *excluding the pairs that cannot start cycles,* e_{ns} *– calls to* equiv *without pre-sorting with memoization,* e_s *– calls to* equiv *with pre-sorting and memoization,* e_{nm} *– call to* equiv *without presorting and without full memoization*

	$\|Q\|$	$\|\delta\|$	cl	min	S_o	S_n	e_{ns}	e_s	e_{nm}
g11	17	34	3	3	6	8	36	33	72
g13	337	673	4	5	32	32	726	692	$63 \cdot 10^6$
g14	137	273	4	5	16	16	297	283	9 852
g15	35	69	4	5	9	9	77	71	283
g4	103	206	2	34	17	20	867	867	3 911
g5	22	42	6	8	3	3	94	46	95
g5p	20	30	10	19	4	4	170	47	172
g6	49	98	2	45	4	4	1 297	1 297	1 318
g7	39	78	2	7	2	2	129	129	132
g8	288	1624	9	16	4	4	2 444	1 636	3 247
g9	232	463	4	5	14	14	482	478	3 407
g9a	16	32	3	3	5	7	33	30	55
griml	17	48	2	2	2	2	55	55	53
java	112	424	19	26	11	18	950	402	9 825
java16	3 186	12 077	19	46	94	94	13 507	12 267	too big
java19	1 971	24 633	24	26	129	129	26 295	24 630	too big
ovis4n	133	613	6	12	14	14	673	637	23 768
ovis5n	44	169	6	12	8	8	194	171	725
ovis6n	17	81	4	5	3	3	77	71	171
ovis7n	13	49	4	4	3	3	44	40	84
ovis9p	2 478	7 434	2	7	27	27	8 239	8 239	27 455
rene2	844	1380	47	193	10	24	40 969	16 434	31 726

Table 2. *Execution times for real-life automata*

	2D	tree	no sort	no mem		2D	tree	no sort	no mem
g13	4	4	3	27 298	ovis4n	2	1	1	12
java16	166	73	262	too big	ovis9p	89	109	81	78
java19	166	73	262	too big	rene2	22	36	28	28

number of classes, but they were not minimal. The ratio of the number of states in the non-minimized automaton to the number of states in the minimized automaton was fixed.

8.2 Ignoring states that cannot start cycles

This improvement relies on density of automata to be minimized, and more precisely on the proportion of states with less than two in-transitions. In all sets of data, such

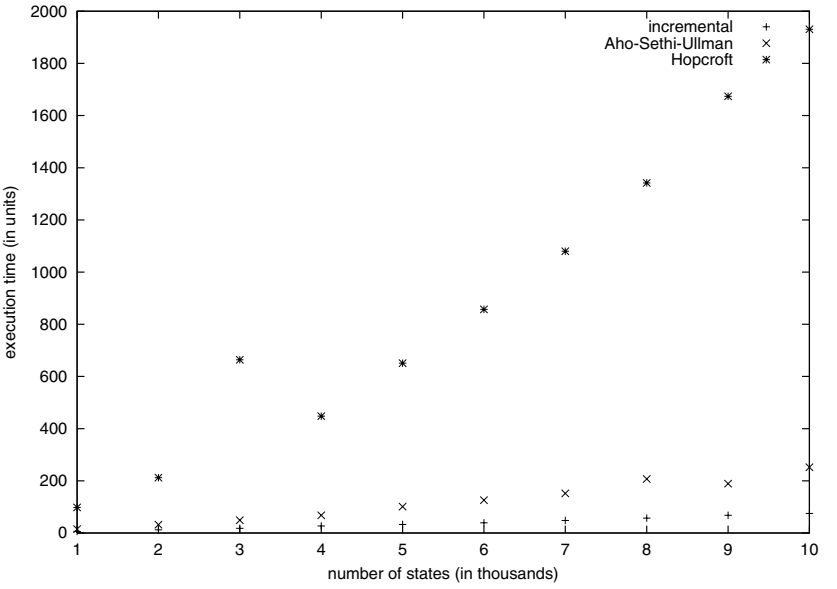

Fig. 3. Execution times for large automata with a large number of classes.

states were not abundant. We have also measured only the longest sequence put into the stack S; perhaps measuring the average length could give different results.

Results for our data show some improvement, but it is not much. Only in one case, our method produced a stack 41.7% of its original size (10 instead of 24). However, even small improvement is worth considering, and there may exist applications where states with fewer than two in-transitions are more common.

8.3 Pre-sorting

Introduction of pre-sorting decreased the number of calls to function equiv in a noticeable way. The average improvement was 16.95% for the first data set. As expected, the biggest improvements were for automata with a large number of classes (see figure 3). Pre-sorting was beneficial in all cases except for automata with only a few classes.

8.4 Full memoization

The original algorithm used only partial memoization. When comparison of a pair of states was inconclusive, the result was lost, and the same pair could be evaluated many times. This lead directly to exponential times for some automata.

Because of the exponential complexity of the original algorithm, its performance was measured only for moderate-size real-life automata. Execution times for the biggest among the small real-life automata are shown in Table 2. Results for large minimal automata with a limited number of classes (figure 4, and results for non-minimal large automata with limited number of classes (figure 5) show that the execution time grows more slowly than the worst case computational complexity.

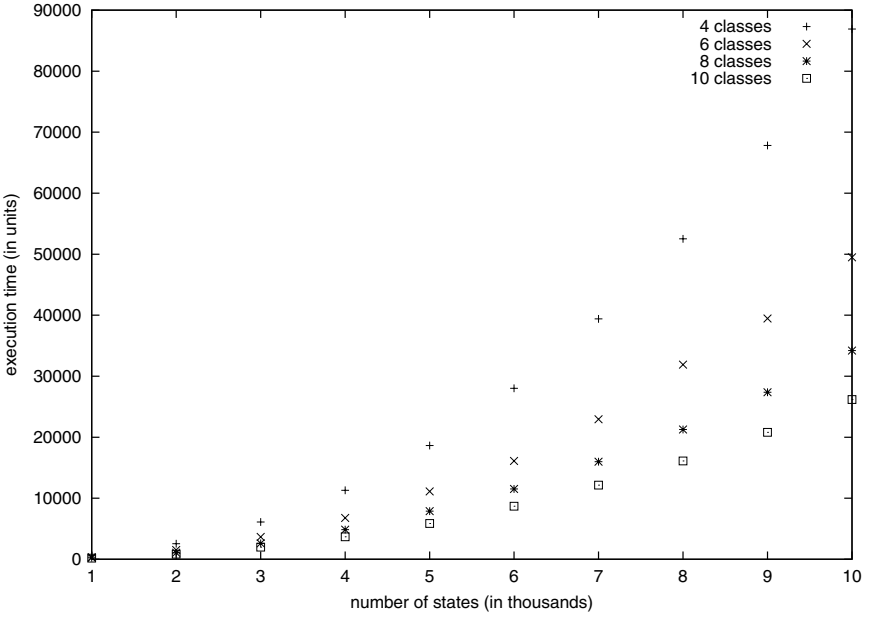

Fig. 4. Execution times for large minimal automata with limited number of classes.

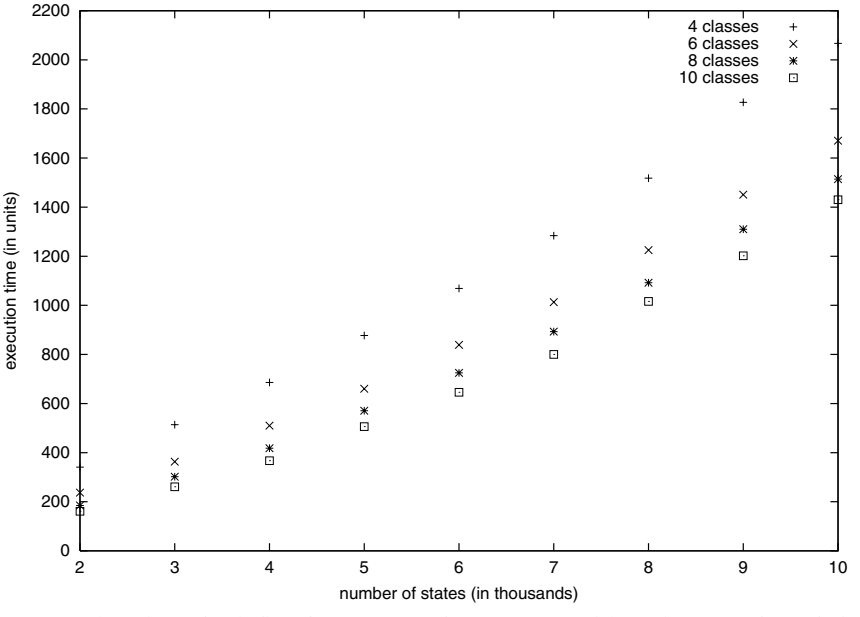

Fig. 5. Execution times for inflated automata of 1000 states with various number of classes.

9 Closing comments

This algorithm has a significant advantage over all of the known algorithms: although function equiv computes only equivalence of pairs of states, the main program computes the entire equivalence relation in such a way that any intermediate

result *Def_Equiv* is usable in (at least partially) reducing the size of an automaton. All of the other known algorithms have unusable intermediate results. This property can be used to reduce the size of automata when the running time of the minimization algorithm is restricted for some reason.

Acknowledgements

We would like to thank the anonymous referees for providing valuable feedback, and Nanette Saes for proofreading this paper. Bruce Watson's original research was supported by Ribbit Software Systems, Inc., and included valuable feedback from Kees Hemerik, Derrick Kourie, Frans Kruseman Aretz, Huub ten Eikelder, Richard Watson, and Gerard Zwaan. Jan Daciuk's research was carried out within the framework of the PIONIER Project *Algorithms for Linguistic Processing*, funded by the NWO (Dutch Organization for Scientific Research) and the University of Groningen. Real test data was provided by Gertjan van Noord, and originally comes from Nederhof (2000). Dale Gerdemann suggested the UNION-FIND algorithm. Sheng Yu suggested bucket sort. The software used in the experiments is available from `http://www.pg.gda.pl/~jandac/minim.html`.

References

Ullman, A. V., Hopcroft, J. E. and Ullman, J. D. (1974) *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

ten Eikelder, H. M. M. (1991) Some algorithms to decide the equivalence of recursive types. *Technical Report 31*, Faculty of Computing Science, Eindhoven University of Technology, The Netherlands.

Hopcroft, J. E. (1971) *An $n \log n$ Algorithm for Minimizing the States in a Finite Automaton*, pp. 189–196. Academic Press.

Hopcroft, J. E. and Ullman, J. D. (1979) *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Nederhof, M.-J. (2000) Practical experiments with regular approximation of context-free languages. *Computational Linguistics* **26**(1): 17–44.

van Noord, G. (2000) The treatment of epsilon moves in subset construction. *Computational Linguistics* **26**(1): 17–44.

Watson, B. W. (1995) Taxonomies and toolkits of regular language algorithms. PhD thesis, Faculty of Computing Science, Eindhoven University of Technology, The Netherlands.

Watson, B. W. (2001) An incremental DFA minimization algorithm. In: Karttunen, L., Koskenniemi, K. and van Noord. G. (eds), *Proceedings Second International Workshop on Finite State Methods in Natural Language Processing*, Helsinki, Finland.

Wood, D. (1987) *Theory of Computation*. Harper & Row.