An incremental DFA minimization algorithm

Bruce W. Watson

Department of Computer Science, University of Pretoria
Pretoria 0002, South Africa
watson@OpenFIRE.org, www.OpenFIRE.org

Faculty of Computing Science
Technical University of Eindhoven
watson@win.tue.nl

Ribbit Software Systems Inc. & FST Labs
watson@fst-labs.com

Abstract

In this paper, we present a new deterministic finite automata 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.

Key words: minimization, deterministic finite automata, incremental algorithms

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 [1].

It computes the equivalence of a given pair of states. It therefore draws upon some non-automata related techniques, such as: structural equivalence of types and memoization of functional programs.

This paper is structured as follows:

- §1.1 gives the mathematical preliminaries required for this paper.
- §2 gives a characterization of minimality of a DFA.
- §3 gives a one-point algorithm which determines whether two states are equivalent.
- §4 gives the incremental algorithm, making use of the one-point algorithm.
- §5 gives the closing comments for the paper.

An early version of this algorithm was presented in [1, §7.4.6].

1.1 Mathematical preliminaries

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.
- $\delta \in Q \times \Gamma \longrightarrow Q \cup \{\perp\}$ 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.
- $F \subseteq Q$ is the set of *final* states.

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

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

Phrased differently, a word z is in $\overrightarrow{\mathcal{L}}(q)$ if and only if

- z is of the form az' where $a \in \Gamma$ is a label of an out-transition from q to $\delta(q,a)$ (ie. $a \in \Gamma_q$) 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:

$$Equiv(p,q) \equiv \overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q)$$

With the inductive definition of $\overrightarrow{\mathcal{L}}$, we can begin rewriting *Equiv* as follows:

$$\begin{array}{ll}
& Equiv(p,q) \\
\equiv & \quad \langle \text{definition of } Equiv \rangle \\
& \overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q) \\
\equiv & \quad \langle \text{the inductive definition of } \overrightarrow{\mathcal{L}} \rangle \\
& (\varepsilon \in \overrightarrow{\mathcal{L}}(p) \equiv \varepsilon \in \overrightarrow{\mathcal{L}}(q)) \wedge \Gamma_p = \Gamma_q \wedge \\
& (\forall\ a : a \in \Gamma_p \cap \Gamma_q : \{a\}\overrightarrow{\mathcal{L}}(\delta(p,a)) = \{a\}\overrightarrow{\mathcal{L}}(\delta(q,a))) \\
\equiv & \quad \langle \text{the inductive definition of } \varepsilon \in \overrightarrow{\mathcal{L}}(p) \rangle \\
& (p \in F \equiv q \in F) \wedge \Gamma_p = \Gamma_q \wedge \\
& (\forall\ a : a \in \Gamma_p \cap \Gamma_q : \{a\}\overrightarrow{\mathcal{L}}(\delta(p,a)) = \{a\}\overrightarrow{\mathcal{L}}(\delta(q,a))) \\
\equiv & \quad \langle \text{for two languages } L_0, L_1\text{: } (\{a\}L_0 = \{a\}L_1) \equiv (L_0 = L_1) \rangle \\
& (p \in F \equiv q \in F) \wedge \Gamma_p = \Gamma_q \wedge \\
& (\forall\ a : a \in \Gamma_p \cap \Gamma_q : \overrightarrow{\mathcal{L}}(\delta(p,a)) = \overrightarrow{\mathcal{L}}(\delta(q,a))) \\
\equiv & \quad \langle \text{definition of } Equiv \rangle \\
& (p \in F \equiv q \in F) \wedge \Gamma_p = \Gamma_q \wedge \\
& (\forall\ a : a \in \Gamma_p \cap \Gamma_q : Equiv(\delta(p,a), \delta(q,a)))
\end{array}$$

All of the algorithms presented in this paper are in the form of the guarded command language — see [2,3].

2 Minimality of DFA*s*

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 [4, §3.4]), minimality

can also be written as the following predicate:

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

(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, and we ignore that issue in the rest of 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 $Equiv(p, q) \equiv$

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

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 [1, Chapter 7]. The pointwise algorithm presented here computes it from below (the *safe* side).

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

From 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 cyclic automata. In order for the functional program to work, it takes a third parameter along with the two states.

The following program computes relation *Equiv* pointwise[1]. An 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 in S, the third parameter) until shown otherwise.

Algorithm 3.1:

[1] It is similar to the one presented in [5]. The algorithm in that technical report computes structural equivalence of types in programming languages.

func $\mathsf{equiv}(p, q, S) \rightarrow$
  if $\{p, q\} \in S \rightarrow eq := true$
  $[\!]$ $\{p, q\} \notin S \rightarrow$
    $eq := (p \in F \equiv q \in F) \wedge (\Gamma_p = \Gamma_q)$;
    $eq := eq \wedge (\forall\ a : a \in \Gamma_p \cap \Gamma_q : \mathsf{equiv}(\delta(p, a), \delta(q, a), S \cup \{\{p, q\}\}))$
  fi;
  return eq
cnuf

□

The $\forall$ quantification can be implemented using a repetition

Algorithm 3.2:

---

func $\mathsf{equiv}(p, q, S) \rightarrow$
  if $\{p, q\} \in S \rightarrow eq := true$
  $[\!]$ $\{p, q\} \notin S \rightarrow$
    $eq := (p \in F \equiv q \in F) \wedge (\Gamma_p = \Gamma_q)$;
    for $a : a \in \Gamma_p \cap \Gamma_q \rightarrow$
      $eq := eq \wedge \mathsf{equiv}(\delta(p, a), \delta(q, a), S \cup \{\{p, q\}\})$
    rof
  fi;
  return eq
cnuf

□

The correctness of the above program can be shown by extending the correctness argument given in [5]. 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 [1, §7.3.3] and [6], it is known that the depth of recursion can be bounded by $(|Q| - 2)$ **max** 0 without affecting the result. To track the recursion depth, we add a parameter k to function equiv such that an invocation $\mathsf{equiv}(p, q, \emptyset, (|Q| - 2)$ **max** $0)$ returns $Equiv(p, q)$. The new function is

Algorithm 3.3:

---

func $\mathsf{equiv}(p, q, S, k) \rightarrow$
  if $k = 0 \rightarrow eq := (p \in F \equiv q \in F)$
  $[\!]$ $k \neq 0 \wedge \{p, q\} \in S \rightarrow eq := true$
  $[\!]$ $k \neq 0 \wedge \{p, q\} \notin S \rightarrow$

$eq := (p \in F \equiv q \in F) \wedge (\Gamma_p = \Gamma_q)$;
for $a : a \in \Gamma_p \cap \Gamma_q \rightarrow$
$eq := eq \wedge \mathsf{equiv}(\delta(p,a), \delta(q,a), S \cup \{\{p,q\}\}, k-1)$
rof
fi;
return eq
cnuf

□

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 [5]. 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 3.4 (Pointwise computation of E):

func $\mathsf{equiv}(p, q, k) \rightarrow$
if $k = 0 \rightarrow eq := (p \in F \equiv q \in F)$
$[\!]$ $k \neq 0 \wedge \{p,q\} \in S \rightarrow eq := true$
$[\!]$ $k \neq 0 \wedge \{p,q\} \notin S \rightarrow$
$eq := (p \in F \equiv q \in F) \wedge (\Gamma_p = \Gamma_q)$;
$S := S \cup \{\{p,q\}\}$;
for $a : a \in \Gamma_p \cap \Gamma_q \rightarrow$
$eq := eq \wedge \mathsf{equiv}(\delta(p,a), \delta(q,a), k-1)$
rof;
$S := S \setminus \{\{p,q\}\}$
fi;
return eq
cnuf

□

The procedure equiv can be memoized[2] to further improve the running time in practice. This algorithm does not appear in the literature.

[2] 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.

Despite the fact that the depth of recursion in Algorithm 3.4 is $(|Q|-2)$ **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 [1].

4 The incremental algorithm

This latest version of function equiv can be used to compute *Equiv*. In variable G, we maintain the pairs of states known to be inequivalent (*distinguished*), while in H, we accumulate our computation of *Equiv*. To initialize G and H, 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 H is transitive at each step[3]. Finally, we have global variable S used in Algorithm 3.4):

Algorithm 4.1 (Computing *Equiv*):

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

□

[3] Since H is initially the identity relation on states, it is already reflexive.

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

Although this algorithm has far worse running time (it is exponential) than the $\mathcal{O}(|Q|\log|Q|)$ of Hopcroft's algorithm [7,8], in practice the incremental algorithm performs acceptably — see [1] where a non-memoizing version of equiv was benchmarked. Presently, it is the only known general incremental minimization algorithm.

5 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 H 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 (for example, in real-time applications).

Acknowledgements: I would like to thank Nanette Y. Saes for proofreading this paper. The two anonymous referees also provided valuable feedback.

References

[1] B. W. Watson, Taxonomies and toolkits of regular language algorithms, Ph.D. thesis, Division of Computer Science, Eindhoven University of Technology, the Netherlands (Sep. 1995).

[2] E. W. Dijkstra, A Discipline of Programming, Prentice Hall, 1976.

[3] D. Gries, The Science of Computer Programming, 2nd Edition, Springer-Verlag, 1980.

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

[5] H. M. M. ten Eikelder, Some algorithms to decide the equivalence of recursive types, Tech. Rep. 31, Division of Computer Science, Eindhoven University of Technology, the Netherlands (1991).

[6] D. Wood, Theory of Computation, Harper & Row, 1987.

[7] J. E. Hopcroft, An $n \log n$ algorithm for minimizing the states in a finite automaton, Academic Press, 1971, pp. 189–196.

[8] D. Gries, Describing an algorithm by Hopcroft, Acta Informatica 2 (1973) 97–109.