ELSEVIER

Information Processing Letters 57 (1996) 65–69

Information Processing Letters

An $O(n \log n)$ implementation of the standard method for minimizing n-state finite automata

Norbert Blum [1]

Informatik IV, Universität Bonn, Römerstr. 164, D-53117 Bonn, Germany

Received 25 April 1994; revised 22 November 1995
Communicated by H. Ganzinger

Abstract

More than 20 years ago, Hopcroft (1971) has given an algorithm for minimizing an n-state finite automaton in $O(kn \log n)$ time where k is the size of the alphabet. This contrasts to the usual $O(kn^2)$ algorithms presented in text books. Since Hopcroft's algorithm changes the standard method, a nontrivial correctness proof for its method is needed. In lectures given at university, mostly the standard method and its straightforward $O(kn^2)$ implementation is presented. We show that a slight modification of the $O(kn^2)$ implementation combined with the use of a simple data structure composed of chained lists and four arrays of pointers (essentially the same as Hopcroft's data structure) leads to an $O(kn \log n)$ implementation of the standard method. Thus, it is possible to present in lectures, with a little additional amount of time, an $O(kn \log n)$ time algorithm for minimizing n-state finite automata.

Keywords: Algorithms; Finite automata; Minimization

1. The standard method

We assume that the reader is familiar with the elementary theory of finite automata as written in standard text books, e.g. [2,3,5,9]. First, we will review the notations used in the subsequence.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a *finite automaton*, where Q is a finite set of states, Σ is a finite, nonempty set of input symbols, δ is a transition function mapping $Q \times \Sigma$ to Q, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of finite states. We extend δ to $Q \times (\Sigma \cup \{\varepsilon\})$ by the arrangement $\delta(q, \varepsilon) = q$ for all $q \in Q$. For $S \subseteq Q$ we define $\delta(S, a) = \bigcup_{q \in S} \delta(q, a)$. Two finite automata are *equivalent*, if they accept the same language. M is *minimal* if no equivalent finite automaton has fewer states than M. Given a finite automaton M, our goal is to compute an equivalent minimal finite automaton. A summary of the known finite automata minimization algorithms is given in [8].

The standard method for the computation of an equivalent minimal finite automaton from a given finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ dues to Huffman and Moore [6,7]. The usual implementation as given in text books uses $O(kn^2)$ time, where $n = |Q|$ and $k = |\Sigma|$. Next we will sketch the standard method as described for instance in [2,3,9].

We say that $x \in \Sigma^*$ *distinguishes* $q, p \in Q$ if $\delta(q, x) \in F$ and $\delta(p, x) \notin F$, or vice versa. We say that q is *distinguishable* from p if there exists $x \in \Sigma^*$ which distinguishes q and p. If q and p are not distin-

[1] Email: blum@cs.uni-bonn.de.

SSDI 0020-0190(95)00199-9

guishable, then q and p are *equivalent*, written $q \equiv p$.

A state $q \in Q$ is *inaccessible* if there is no $x \in \Sigma^*$ with $\delta(q_0, x) = q$. By the Myhill–Nerode theorem (see [5]), a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is minimal if and only if no state in Q is inaccessible and no two distinct states in Q are equivalent. It is easy to delete all inaccessible states from a finite automaton in linear time (see Algorithm 0.3 in [2]). Hence, we can assume that in the given finite automaton M, no states are inaccessible.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton with no inaccessible states. Then, the computation of an equivalent minimal finite automaton is reduced to the computation of the equivalence classes of Q with respect to the relation $\equiv$, followed by the straight forward modification of the transition function δ. By definition, we obtain

$$q \not\equiv p \quad \text{iff} \quad \text{there is } a \in \Sigma\colon \delta(q,a) \not\equiv \delta(p,a).$$

This led to the following standard method.

Algorithm $DFA \rightsquigarrow DFA_{\min}$

Input: A finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ with no inaccessible states.

Output: A minimal finite automaton $M' = (Q', \Sigma, \delta', q_0', F')$.

Method:

(1) $t := 2$; $Q_0 := \{\textit{undefined}\}$; $Q_1 := F$; $Q_2 := Q \setminus F$.

(2) **while** there is $0 < i \leqslant t$, $a \in \Sigma$ with $\delta(Q_i, a) \not\subseteq Q_j$, for all $j \leqslant t$
do
1. Choose such an i, $a \in \Sigma$, and $j \leqslant t$ with $\delta(Q_i, a) \cap Q_j \neq \emptyset$.
2. $Q_{t+1} := \{q \in Q_i \mid \delta(q,a) \in Q_j\}$;
$Q_i := Q_i \setminus Q_{t+1}$;
$t := t + 1$.

od.

(3) (* Let $[q]$, $q \in Q$ be the equivalence class Q_i with $q \in Q_i$. *)

$Q' := \{Q_1, Q_2, \ldots, Q_t\}$.

$q_0' := [q_0]$.

$F' := \{[q] \in Q' \mid q \in F\}$.

$\delta'([q], a) := [\delta(q,a)]$ for all $q \in Q$, $a \in \Sigma$.

It is easy to see that the time complexity of the straight forward implementation of the standard method is $O(kn^2)$. Note that the initialization of the algorithm can be generalized to any starting equivalence relation or its partition, respectively. Hence we can minimize with the standard method finite automata with distinguished finite states, too. Hopcroft's algorithm is presented within this general framework in [1].

2. An $O(kn \log n)$ implementation of the standard method

Step 2 is the only part of the algorithm which is not easily implemented in linear time. The straight forward implementation for checking the condition of the while loop, and for the execution of the body of the while loop is time consuming. Hence, $O(kn^2)$ time is needed in the worst case. For speeding up the execution of the body of the while loop, we will modify the body slightly such that we can ensure for all $q \in Q$ that the name of the class which contains q changes at most $\log n$ times,

Modification of the body of the while loop:

1. Choose such an i, $a \in \Sigma$, and choose $j_1, j_2 \leqslant t$ with $j_1 \neq j_2$, $\delta(Q_i, a) \cap Q_{j_1} \neq \emptyset$, and $\delta(Q_i, a) \cap Q_{j_2} \neq \emptyset$.
2. **If** $|\{q \in Q_i \mid \delta(q,a) \in Q_{j_1}\}| \leqslant |\{q \in Q_i \mid \delta(q,a) \in Q_{j_2}\}|$
then $Q_{t+1} := \{q \in Q_i \mid \delta(q,a) \in Q_{j_1}\}$
else $Q_{t+1} := \{q \in Q_i \mid \delta(q,a) \in Q_{j_2}\}$ **fi**;
$Q_i := Q_i \setminus Q_{t+1}$;
$t := t + 1$.

By the choice of i, it is clear that j_1, j_2 exists. Note that with respect to Q_i, before the definition of Q_{t+1},

$$|Q_{t+1}| \leqslant 1/2|Q_i|.$$

Hence, for all $q \in Q$ the name of the class which q contains changes at most $\log n$ times. Our goal is to develop an implementation such that all work can be assigned to transitions containing a state for which the name of the corresponding class is changed. Next we will introduce a data structure which enables an $O(kn \log n)$ implementation of the modified Step 2.

This data structure has to support the following operations:

1. The choice of $i \leqslant t$, $a \in \Sigma$ with $\delta(Q_i, a) \not\subseteq Q_j$ for all $j \leqslant t$;
2. the choice of $j_1, j_2 \leqslant t$ with $j_1 \neq j_2$, $\delta(Q_i, a) \cap Q_{j_1} \neq \emptyset$, and $\delta(Q_i, a) \cap Q_{j_2} \neq \emptyset$;
3. the decision if $|\{q \in Q_i \mid \delta(q, a) \in Q_{j_1}\}| \leqslant |\{q \in Q_i \mid \delta(q, a) \in Q_{j_2}\}|$; and
4. the construction of Q_{t+1}.

Let $\mathcal{D}_t$ denote the data structure obtained directly after the construction of Q_t, $t \geqslant 2$. Essentially, $\mathcal{D}_t$ coarsen the transition function δ to $\delta' \subseteq [1..t] \times \Sigma \times [1..t]$, defined by

$$(i, a, j) \in \delta' \quad \text{iff} \quad \text{there is } q \in Q_i,\ p \in Q_j \text{ with } \delta(q, a) = p.$$

For each triple $(i, a, j) \in \delta'$, $\mathcal{D}_t$ contains a doubly chained list $L(i, a, j)$ which contains exactly those states q in Q_i with $\delta(q, a)$ is in Q_j. Each element in the list has an additional pointer to the head of the list which contains all needed information. The size of the list $L(i, a, j)$ is stored in the variable $S(i, a, j)$ such that the size can be checked in constant time. Additionally, we have a $|Q| \times |\Sigma|$-array Δ and a $|\Sigma| \times |Q|$-array Δ^{-1}. The component $\Delta(p, a)$, $p \in Q, a \in \Sigma$ contains a pointer to the unique record, containing p with respect to the lists $L(\cdot, a, \cdot)$. Note that the finite automaton is deterministic. The component $\Delta^{-1}(b, q)$, $b \in \Sigma, q \in Q$ contains a pointer to a list containing exactly those states $p \in Q$ with $\delta(p, b) = q$. We will identify this set of states with $\Delta^{-1}(b, q)$. For each pair (i, a), $i \in [1..t]$, $a \in \Sigma$, we update a list $\Delta'(i, a)$, containing pointers to those $L(i, a, j)$ with $(i, a, j) \in \delta'$. The head of list $L(i, a, j)$ contains a pointer to the element in list $\Delta'(i, a)$ which points to $L(i, a, j)$. In a set K, we maintain pointers to those lists $\Delta'(i, a)$ of length $\geqslant 2$. The head of $\Delta'(i, a)$ contains a pointer to the element of K which points to $\Delta'(i, a)$. The data structure is illustrated by Fig. 1. In addition to $\mathcal{D}_t$, we need two arrays Γ and Γ' for the efficient computation of $\mathcal{D}_{t+1}$.

It is easy to see that at the beginning, the data structure $\mathcal{D}_2$ can be constructed in $O(kn)$ time. Given the structure $\mathcal{D}_t$, we will describe how to perform Step 2 with respect to the construction of $\mathcal{D}_{t+1}$.

(i) Use the set K for obtaining a list $\Delta'(i, a)$ of length $\geqslant 2$ in constant time, and choose any two triples $(i, a, j_1), (i, a, j_2)$ in $\Delta'(i, a)$. With help of $S(i, a, j_1)$ and $S(i, a, j_2)$ decide if $|L(i, a, j_1)| \leqslant |L(i, a, j_2)|$. Let $L(i, a, j_{\min})$ denote the shorter list. Change the first component of the triple $(i, a, j_{\min})$ from i to $t+1$. Now, the old list $L(i, a, j_{\min})$ is the list $L(t+1, a, j_{\min})$, and hence, $S(t+1, a, j_{\min})$ is equal to $S(i, a, j_{\min})$. Delete the pointer to $L(i, a, j_{\min})$ from $\Delta'(i, a)$, and insert a pointer which points now to $L(t+1, a, j_{\min})$ into a new list $\Delta'(t+1, a)$. If the length of the list $\Delta'(i, a)$ is now smaller than 2, then delete the pointer to $\Delta'(i, a)$ from K.

(ii) For all q in $L(t+1, a, j_{\min})$, we update the data structure in the following way:

(1) For all $b \in \Sigma \setminus \{a\}$, we use $\Delta(q, b)$ for finding the unique record which contains q with respect to the lists $L(i, b, \cdot)$, if it exists. Assume that $L(i, b, k)$ is the list containing this record. Then we delete this record from $L(i, b, k)$, inserting it into $L(t+1, b, k)$. For the efficient determination of $L(t+1, b, k)$, we maintain an additional $|\Sigma| \times |Q|$-array Γ. The component $\Gamma(b, k)$ contains a pointer to the last generated list $L(j, b, k)$. If $j \neq t+1$, then we have to generate a new list. $S(i, b, k)$ is decreased by 1, and $S(t+1, b, k)$ is increased by 1. If $L(i, b, k)$ becomes empty, we delete the pointer to $L(i, b, k)$ from $\Delta'(i, b)$ and eventually, the pointer to $\Delta'(i, b)$ from K. In the case that $L(t+1, b, k)$ gets defined, we have to add to $\Delta'(t+1, b)$ a pointer to $L(t+1, b, k)$, and eventually, we have to add to K a pointer to $\Delta'(t+1, b)$. $\Gamma(b, k)$ is modified appropriately.

(2) For all $b \in \Sigma$, for all $p \in \Delta^{-1}(b, q)$, we use $\Delta(p, b)$ for finding the unique record which contains p with respect to the lists $L(\cdot, b, i)$, if it exists. Assume that $L(k, b, i)$ contains p. Then we delete p from this list, inserting it into $L(k, b, t+1)$. Analogously to above, we maintain an additional $|Q| \times |\Sigma|$-array Γ' such that $L(k, b, t+1)$ can be determined in constant time. $S(k, b, i)$ is decreased by 1, and $S(k, b, t+1)$ is increased by 1. If $L(k, b, i)$ becomes empty, we delete the pointer to $L(k, b, i)$ from $\Delta'(k, b)$ and eventually, the pointer to $\Delta'(k, b)$ from K. In the case that $L(k, b, t+1)$ gets defined, we have to add to $\Delta'(k, b)$ a pointer to $L(k, b, t+1)$, and eventually, we have to add to K a pointer to $\Delta'(k, b)$. $\Gamma'(k, b)$ is modified appropriately.

By construction, it is easy to see that the structure $\mathcal{D}_{t+1}$ is correctly computed. Note that after the termi-

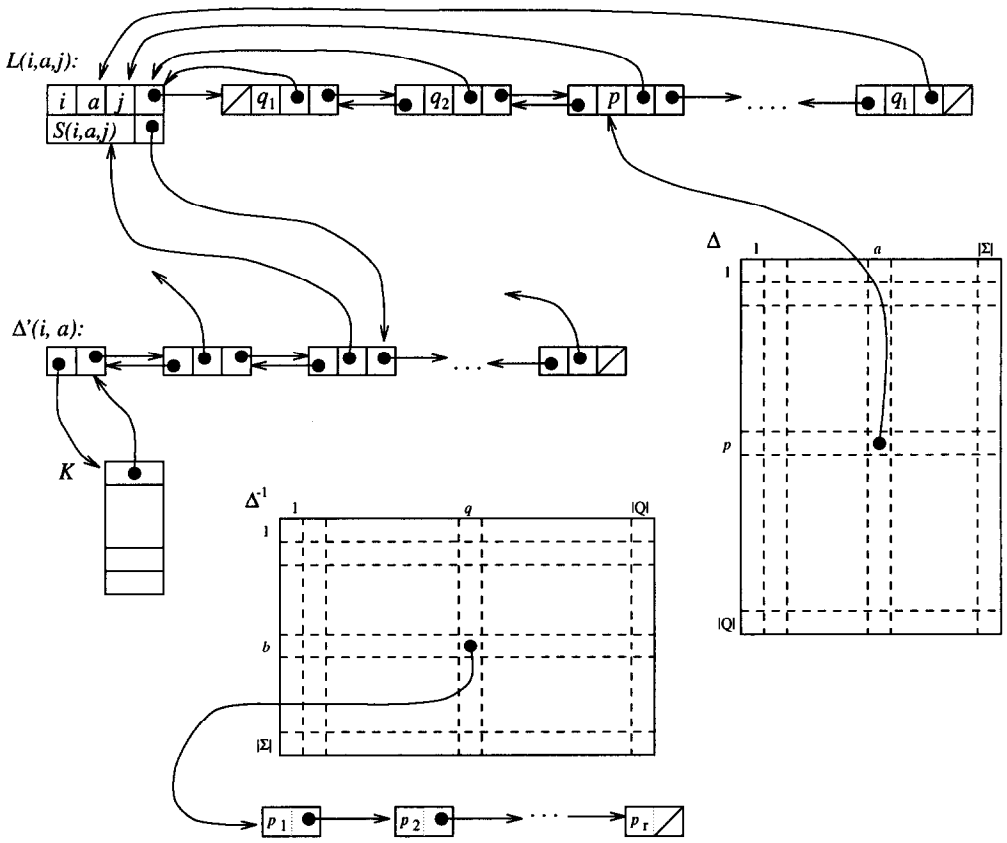

Fig. 1.

nation of the algorithm, the resulting δ' is the transition function of the minimal finite automaton. It remains to check the time complexity.

Step (i) needs only constant time. In Step (ii.1) and Step (ii.2), the total time for transferring one record from one list to another list is constant. Every time a record is moved in Step (ii.1), a transition $\delta(q, a) = p$ of the transition function for which q has changed the name of its class is considered. Every time a record is moved in Step (ii.2), a transition $\delta(p, a) = q$ of the transition function for which q has changed the name of its class, is considered. Hence, every time such a transition is considered, at least one of the two states in this transition has changed the name of its class. Hence, each transition in δ is considered at most $2 \log n$ times. There are at most kn transitions in δ. Hence, the total time used for Step (ii) is bounded by $O(kn \log n)$. Altogether we have obtained the following theorem.

Theorem 1. *The standard method for minimizing an n-state finite automaton can be implemented such that the used time is* $O(kn \log n)$.

Acknowledgment

I thank Burchard von Braunmühl and Claus Rick for critical remarks.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974) 157–162.

[2] A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation and Compiling, Vol. I: Parsing* (Prentice-Hall, Englewood Cliffs, NJ, 1972).

[3] W. Brauer, *Automatentheorie* (Teubner, Stuttgart, 1984).

[4] J.E. Hopcroft, An $n \log n$ algorithm for minimizing the states in a finite automaton, in: Z. Kohavi, ed., *The Theory of Machines and Computations* (Academic Press, New York, 1971) 189–196.

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

[6] D.A. Huffman, The synthesis of sequential switching circuits, *J. Franklin Institute* **257** (1954) 3–4, 161–190, 275–303.

[7] E.F. Moore, Gedanken experiments on sequential machines, in: C.E. Shannon and J. McCarthy, eds., *Automata Studies* (Princeton University Press, Princeton, NJ, 1956) 129–153.

[8] B.W. Watson, A taxonomy of finite automata minimization algorithms, Computing Science Rept. 93/44, Eindhoven University of Technology, The Netherlands, 1993.

[9] D. Wood, *Theory of Computation* (Harper & Row, New York, 1987).