Department of Computer Science,
University of Otago

Technical Report OUCS-2004-11

A Practical Algorithm for Reducing Non-deterministic Finite State Automata

Authors:
M. H. Albert
Department of Computer Science, University of Otago
S. Linton
Centre for Interdisciplinary Research in Computational Algebra,
University of St. Andrews

Status: Submitted to *Theoretical Computer Science*

Department of Computer Science,
University of Otago, PO Box 56, Dunedin, Otago, New Zealand

http://www.cs.otago.ac.nz/research/techreports.html

A Practical Algorithm for Reducing Non-deterministic Finite State Automata

Michael Albert [a,1] Steve Linton [b]

[a] *Department of Computer Science, University of Otago, Dunedin, New Zealand*

[b] *Centre for Interdisciplinary Research in Computational Algebra, University of St Andrews, Fife, Scotland*

Abstract

In [3], Ilie and Yu describe a construction of a right-invariant equivalence relation on the states of a non-deterministic finite-state automaton. We give a more efficient algorithm for constructing the same equivalence, together with results from a computer implementation.

---

1 Introduction

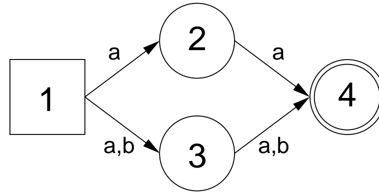
Finite state automata (FSAs) are ubiquitous in many areas of computer science. They are the most computationally useful representation of the class of regular languages, with applications in parsing computer languages and text manipulation, and also of mathematical interest through connections to the theory of transformation monoids, to geometric group theory and to combinatorics.

Most constructions of FSAs naturally produce non-deterministic automata (NFAs) and, while equivalent deterministic automata (DFAs) can be produced, they may be exponentially larger. Nevertheless, this conversion is often required in practice, and one reason for this is that there is an essentially unique minimal DFA recognising any regular language, which can be efficiently constructed from any DFA for the language. Thus, a common pattern in calculations with FSAs is to begin with a NFA, construct a possibly much larger DFA that accepts the same language and then construct the, hopefully much smaller, minimal DFA from this.

[1] This work was supported by EPSRC grant GR/S41074/01

Preprint submitted to Elsevier Science *15 July 2004*

Fig. 1. A non-deterministic automaton which cannot be reduced



In many treatments, NFAs can have ϵ-transitions which consume no input symbol. Standard and efficient algorithms exist to convert NFAs with ϵ-transitions into NFAs with the same number of states and no ϵ-transitions. We will assume that this has been done if necessary and that our NFAs have no ϵ-transitions.

In this paper we present an algorithm which, given an NFA, computes a possibly smaller NFA recognising the same language. This smaller automaton can be used directly, or it can be taken as a smaller and hopefully more tractable input to the determinisation and minimization procedure outlined above. We give experimental results showing that this can be extremely effective.

There is, in general, no unique minimal NFA equivalent to a given one in the same way that there is a unique minimal DFA. Nevertheless, the output of our algorithm is minimal in a more limited sense. Specifically, we compute (the quotient automaton by) the coarsest *right-invariant* equivalence relation on the set of states of the input automaton. This is an especially appropriate choice if the result is to be input to the determinisation algorithm, as that will effectively quotient out any left-invariant equivalence as a part of its working.

It is instructive to consider a simple example which illustrates why this method does not provide a unique minimal NFA. The four state automaton N shown in Figure 1 recognises the language $\{a, b\}\{a, b\}$ and is equivalent to the three state automaton obtained by deleting state 2 and its associated transitions. However, the languages of words which take it from the initial state to each of its states are all different, as are the languages accepted beginning in each state, so that there are no non-trivial left or right invariant equivalence relations.

When we determinise this automaton, the DFA has states corresponding to the following sets of states of N: $\{1\}$, $\{2, 3\}$, $\{3\}$ and $\{4\}$. Now the languages accepted from states $\{2, 3\}$ and $\{3\}$ are the same, and these states will be identified in the minimization algorithm, resulting in a minimal DFA with 3 states.

2 Notation

We consider as input an NFA N given by a quintuple (Q, A, δ, q_0, F) where Q is the set of states, A the alphabet, $\delta : Q \times A \to \mathcal{P}(Q)$ the transition function, $q_0 \in Q$ is the starting state and $F \subseteq Q$ the set of accepting states. We abuse notation slightly by sometimes treating δ as a relation $\delta \subseteq Q \times A \times Q$, rather than a set-valued function.

We denote the number of elements of Q by n, the number of elements of A by l and the number of elements of δ (construed as a relation) by m.

It is convenient to assume that every state has at least one transition under each symbol. If this is not the case, an equivalent automaton for which it is the case can be computed easily by adjoining a single non-accepting, non-initial state $\emptyset$ and adding transitions to it from every state (including itself) under every symbol for which a transition in N was not defined.

Under this assumption, we observe that $nl \leq m \leq n^2 l$, and N needs $O(m(\log n + \log l))$ bits to specify.

3 The Equivalence Relation

The relation $\equiv_R$ is defined by Ilie and Yu in [3] by forming the coarsest relation satisfying two conditions:

(1) No final state is equivalent to any non-final state
(2) No pair of states p and q can be equivalent if there exists an a and r such p has a transition to r under a and q has no transition under a to any state equivalent to r.

They show that $\equiv_R$ is the coarsest right-invariant equivalence relation.

They give an algorithm to compute $\equiv_R$ but no complexity estimate. A simple-minded analysis suggests a running time of $O(n^5 l)$.

We also construct the relation by a closure process, but we organise the calculation rather more efficiently. To do this we make use the inverse of the transition function δ, and we extend its definition to sets of states in the obvious way

$$\delta^{-1}(C, a) = \bigcup_{c \in C} \delta^{-1}(c, a) = \{q \in Q : \delta(q, a) \cap C \neq \emptyset\}$$

We define our relation $\sim$ to be the coarsest equivalence relation with the following properties:

(1) No final state is equivalent to any non-final state
(2) For any equivalence classes of states C and D and any letter a, either $D \subseteq \delta^{-1}(C, a)$ or $D \cap \delta^{-1}(C, a) = \emptyset$.

The second condition simply states formally the requirement that for each equivalence class of states C and each letter a, the set $\delta^{-1}(C, a)$ is a union of equivalence classes.

Proposition 1 *The two equivalence relations defined above are the same. That is:* $\sim = \equiv_R$

PROOF. We prove that Ilie and Yu's second condition is equivalent to ours. Suppose that we have a relation $\approx$ satisfying their two conditions, but not satisfying our second condition. Let C, a and D be such that $p \in D \setminus \delta^{-1}(C, a)$ and $q \in D \cap \delta^{-1}(C, a)$. Then there must be some $s \in C \cap \delta(q, a)$ while $\delta(p, a) \cap C = \emptyset$ so that $s \not\approx t$ for all $t \in \delta(p, a)$, contradicting Ilie and Yu's second condition.

Conversely, suppose that $\approx$ satisfies our conditions but not Ilie and Yu's. Let p, q, a, r be such that $p \approx q$, $r \in \delta(p, a)$ and for all $s \in \delta(q, a)$, $r \not\approx s$. There are two cases:

- If $r = \emptyset$ then $\emptyset \notin \delta(q, a)$. So choose any element s of $\delta(q, a)$ and let C be its equivalence class. Since, by our construction of the transitions to $\emptyset$, $\delta(p, a) = \{\emptyset\}$ we see that $q \in \delta^{-1}(C, a)$ but $p \notin \delta^{-1}(C, a)$ which contradicts our second condition.
- If $r \neq \emptyset$ then let C be the equivalence class of r. Then $p \in \delta^{-1}(C, a)$ but $q \notin \delta^{-1}(C, a)$ giving the same result.

Since the conditions are equivalent, and both $\sim$ and $\equiv_R$ are defined as coarsest equivalence relations satisfying the respective conditions, they must be equal.

4 The Algorithm

Based on our definition of the equivalence relation, we can give a more efficient algorithm to construct it, which appears as Algorithm 1.

Proposition 2 *Algorithm 1 correctly computes* $\sim$.

```
 1: PENDING := {F, Q \ F}
 2: R := {F, Q \ F}
 3: Precompute δ^{-1} on states
 4: while PENDING is not empty do
 5:    pick a class C from PENDING
 6:    splitC := false
 7:    for a ∈ A while not splitC do
 8:       P := δ^{-1}(C, a)
 9:       for all D in R do
10:          D_1 := D ∩ P
11:          D_2 := D \ P
12:          if D_1 = ∅ or D_2 = ∅ then
13:             continue
14:          end if
15:          Delete D from R and from PENDING (if it is there)
16:          Add D_1 and D_2 to R and to PENDING.
17:          if D = C then
18:             splitC := true
19:          end if
20:       end for
21:    end for
22:    Delete C from PENDING (if it is still there)
23: end while
24: Return R
```

Algorithm 1: Compute the relation $\sim$

PROOF. To understand the algorithm we first introduce a small piece of terminology. We say that a set C *splits* a set D *using the letter* a if both $D \setminus \delta^{-1}(C,a)$ and $D \cap \delta^{-1}(C,a)$ are non-empty. The algorithm begins with the coarsest possible partition of the set of states consistent with the first condition. It maintains a queue, called PENDING, of states for which the second condition has not yet been verified, and also a set R consisting of the current set of equivalence classes. At each iteration of the outer **while** loop (lines 4 through 23) a class C is chosen from PENDING. If C splits any class D of R then the two parts of D formed by the split are added to PENDING and also to R. At the end of this loop, unless C has split itself, it can be deleted from PENDING. Of course, if it is later split by some other set, then the parts will be returned to the PENDING queue.

The invariants of the algorithm are that no two classes in $R \setminus$ PENDING can violate our second condition for any a, that R is a partition of Q and that PENDING is a subset of R. It is easy to see that these are maintained and guarantee correctness.

Termination is guaranteed since either the number of parts in R increases or the number in PENDING decreases at each pass round the outer loop, and the number of elements of R is bounded by n.

Proposition 3 *The running time of Algorithm 1 is is $O(mn)$.*

PROOF. We will assume that images, inverse images and equivalence classes are represented in appropriate data structures, such as hash tables, permitting insertion, deletions and membership testing in $O(1)$ time. Let k be the number of equivalence classes of the relation constructed (i.e. the final size of R). Of course $k \leq n$.

Computing all the inverse images takes time $O(m)$ which is also their total length.

We make at most one pass around the outer **while** loop with every class C that we ever create. These classes can be organised in two binary trees with the classes of $\sim$ at the leaves and F and $Q \setminus F$ at the roots. The total number of nodes in such trees is twice the number of leaves minus 2, i.e. $2k - 2$, and so $O(n)$.

For each C the associated executions of line 8 require up to $O(m)$ time to compute the inverse images of a class by taking unions of the inverse images of the states in it. The total time spent in line 8 is thus $O(km)$.

The splitting of a class D in lines 10 and 11 can be done in time $|D|$, since each point must be looked at to see if it is in P or not, so splitting all classes with a given P takes at most $O(n)$ time. The **for** loop from lines 7 to 21 is entered at most $2kl$ times, so the total time spent in lines 10 and 11 is at most $O(nkl)$.

The adjustment of PENDING and R (lines 15 and 16) can be done in constant time per iteration using (for instance) doubly-linked lists.

Putting together all of these estimates we obtain a running time of $O(km + kln)$. Since $k \leq n$ and $kl \leq nl \leq m$ this gives the $O(mn)$ result claimed above.

Once the equivalence relation is constructed, and known to be right invariant, constructing the quotient automaton is straightforward, taking time $O(m)$.

5 Implementation

We have implemented this algorithm in the GAP language, in the context of a package of tools for computing with finite state automata which is under development. We were motivated by some problems in the theory of closed classes of permutations which led to the construction of quite large (tens of thousands of states) non-deterministic automata, which we needed to determinise and then minimize (see [2] for more details). Experience suggested that the minimal deterministic automata were not too large (comparable in size to the non-deterministic automata) but the size of the intermediate non-minimal deterministic automata was much greater (millions of states) and was the limiting factor in our computations.

We used the algorithm of this paper to reduce the size of the non-deterministic automata, before determinising, thus reducing the size of the intermediate automata. Of course the final, minimal automata are essentially unique so their size is unchanged by this extra step.

The implementation closely followed the algorithm as described. PENDING was represented by a doubly-linked list, and R by a flexible array. When a class D was split, it was replaced in R by the larger of D_1 and D_2, and the smaller was appended to R. An additional data structure recorded for each point, the index in R of its class. This enabled the **forall** loop to run only over those classes that actually contained an element of P, a useful improvement in practice.

5.1 Performance

We applied our implementation to automata describing the classes of permutations generated by a 2-stack and a k-stack in parallel for various values of k. Specifically these are non-deterministic (but ϵ-free) automata recognising the reverse of the rank-encoding of these permutations. The alphabet size for these languages is always $k + 4$. For each automaton, we consider the two routes from the original NFA to a minimal DFA shown in Figure 2.

For each example, Table 1 gives the numbers of states of the automata **A**–**E**, and the number of transitions for the non-deterministic ones, and the run time (in seconds on a 3.06GHz Pentium 4) of the five computations **1**–**5**. Comparing totals **1+2+4** to **3+5** clearly shows the advantage of reducing before determinising.

Fig. 2. Two Routes to a Minimal DFA

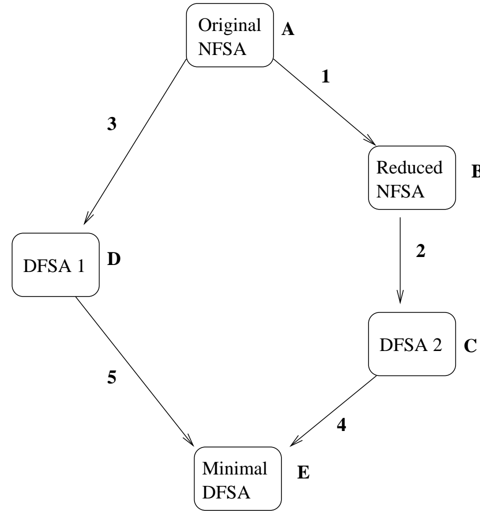


Table 1
Experimental Results

	Automata Sizes							Run Times				
	A		**B**		**C**	**D**	**E**					
k	st	tr	st	tr	st	st	st	**1**	**2**	**3**	**4**	**5**
2	66	1524	6	33	12	42	10	0.00	0.00	0.00	0.00	0.00
3	172	7647	21	202	105	288	98	0.00	0.00	0.06	0.00	0.00
4	432	37K	36	443	374	900	362	0.05	0.02	0.62	0.01	0.02
5	1056	184K	57	859	1047	2299	1027	0.25	0.10	7.39	0.05	0.07
6	2528	888K	85	1527	2583	5272	2549	1.39	0.30	82.78	0.15	0.21
7	5952	4.2M	121	2543	5904	11K	5844	9.08	1.12	857.9	0.44	0.59
8	13K	19M	166	4024	12K	23K	12K	56.61	2.34	9903	1.47	1.90

6 Concluding Remarks

We have given a useful, effective and practical algorithm for simplifying NFAs. The output is the same as that of the algorithm of [3], but, by reformulating the definition of the equivalence relation, we obtain a much faster algorithm.

Our algorithm is extremely similar to the minimization algorithm for DFAs given in [1] §4.13. The only difference is that when a class D is split into D_1 and D_2, that algorithm only has to consider one of these new classes as a possible splitter of other classes (and, of course, they choose the smaller one), whereas we have to consider both. This (together with the fact that $m = nl$ for a DFA) gives them a runtime $O(m \log n)$ to our $O(mn)$.

We remark on the attractive feature of this algorithm that the factor of n in the time complexity is actually an upper bound on a factor of k, the number of states of the output automaton, so that when k is small, and large reductions are possible, they will be found quickly. This motivates the suggestion that in some applications it might be correct to run this algorithm to attempt to reduce an automaton, but to abandon the calculation if it is taking too long, suggesting that little reduction will be achieved.

References

[1] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

[2] M. H. Albert, M. D. Atkinson, and N. Ruškuc. Regular closed sets of permutations. *Theoret. Comput. Sci.*, 306(1-3):85–100, 2003.

[3] Lucian Ilie, Sheng Yu: Reducing NFAs by invariant equivalences. Theor. Comput. Sci. 306(1-3): 373-390 (2003)