

Instruction Computation in Subset Construction[‡]

J. Howard Johnson* and Derick Wood[†]

Technical Report HKUST-CS96-21
June 1996

*Institute for Information Technology
National Research Council
Nepean, Ontario
Canada

[†]Department of Computer Science
Hong Kong University of Science & Technology
Clear Water Bay, Kowloon
Hong Kong

Abstract

Subset construction is *the* method of converting a nondeterministic finite-state machine into a deterministic one. The process of determinization is an important one in any implementation of finite-state machines since nondeterministic machines are often easier to describe than their deterministic equivalents and the conversion of regular expressions to finite-state machines usually produces nondeterministic machines.

We discuss one aspect of subset construction; namely, the computation of the instructions of the equivalent deterministic machine. Although the discussion is to some extent independent of any specific assumptions, we draw some conclusions within the context of INR and Grail, both packages for the manipulation of finite-state machines.

The aim of the discussion is to present the problem and suggest some possible solutions; we do not intend to and cannot be definitive since much remains unknown.

[‡]This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada, from the Information Technology Research Centre of Ontario, and from the Research Grants Committee of Hong Kong.



Instruction Computation in Subset Construction¹

J. Howard Johnson² Derick Wood³

Abstract

Subset construction is *the* method of converting a nondeterministic finite-state machine into a deterministic one. The process of determinization is an important one in any implementation of finite-state machines since nondeterministic machines are often easier to describe than their deterministic equivalents and the conversion of regular expressions to finite-state machines usually produces nondeterministic machines.

We discuss one aspect of subset construction; namely, the computation of the instructions of the equivalent deterministic machine. Although the discussion is to some extent independent of any specific assumptions, we draw some conclusions within the context of INR and Grail, both packages for the manipulation of finite-state machines.

The aim of the discussion is to present the problem and suggest some possible solutions; we do not intend to and cannot be definitive since much remains unknown.

1 Introduction

Subset construction is *the* method of converting a nondeterministic finite-state machine into a deterministic one. The basic idea underlying the conversion is the use of sets of states (state sets) of the nondeterministic finite-state machine as states in the corresponding equivalent deterministic finite-state machine. It is simple, yet nontrivial to implement well. One immediate practical problem is that if the given nondeterministic finite-state machine has n

¹This work was supported under grants from the Natural Sciences and Engineering Research Council of Canada, from the Information Technology Research Centre of Ontario, and from the Research Grants Committee of Hong Kong.

²Institute for Information Technology, National Research Council, Nepean, Ontario, Canada. E-mail: johnson@iit.nrc.ca.

³Department of Computer Science, Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: dwood@cs.ust.hk.

states, the resulting deterministic finite-state machine may have $2^n - 1$ states and there are nondeterministic finite-state machines that yield $\Omega(2^n)$ states. Thus, the exponential growth factor cannot be avoided in general. There are two possible directions, at least, of investigation that this bleak news leads to: to improve the efficiency of determinization under the assumption that exponential blow up is pathological and to analyze the expected size of the resulting deterministic finite-state machines. We follow the first direction here; we explore the second direction in a companion paper [7] based on the preliminary results of Leslie [6].

This abstract raises more questions than it solves since it deals with applied algorithms and applied data structuring. Nevertheless, we believe that our suggestions are of interest in that they may encourage others to tackle some of the remaining questions.

2 Subset Construction

We assume that readers are familiar with the notions of a finite-state machine; however, we recall the basic definitions. A finite-state machine M is specified in a standard manner by a tuple $(Q, \Sigma, \delta, s, F)$, where Q is a finite set of states, Σ is a finite alphabet (the input alphabet), δ is a finite program given as a set of instructions of the form

$$(\text{source-state}, \text{source-symbol}, \text{target-state}),$$

s is the start state of M , and $F \subseteq Q$ is a set of final states. A computation of M on an input string $a_1 \cdots a_m$ corresponds to one executable sequence of instructions

$$(p_1, a_1, p_2)(p_2, a_2, p_3) \cdots (p_m, a_m, p_{m+1}),$$

where $p_1 = s$. If p_{m+1} is in F , then M accepts $a_1 \cdots a_m$. Given a finite-state machine M in which, for every state p and every symbol a , there is at most one instruction of the form $(p, a, ?)$ in δ , then M is deterministic. When M is nondeterministic, there may be many different M -computations on an input string.

It is not difficult to compile a deterministic finite-state machine into a subprogram of a given programming language and even into a VLSI chip. But compiling a nondeterministic finite-state machine is more costly since the compiled code must incorporate backtracking to handle the nondeterminism. An elegant compilation was given by Thompson [9]; it has never

been matched. An alternative and standard approach is to convert a nondeterministic finite-state machine into an equivalent deterministic finite-state machine using subset construction. (An intermediate approach is to do subset construction on the fly for those state sets reached by the input string. We do not discuss this idea any further in this paper.) Wood [10] gave the following algorithm for subset construction (it is clearly not intended for direct implementation):

Algorithm Subset Construction
On entry: A nondeterministic finite-state machine, $M = (Q, \Sigma, \delta, s, F)$.
On exit: A deterministic finite-state machine, $M' = (Q', \Sigma, \delta', s', F')$
that satisfies $L(M) = L(M')$.
begin
Let $Q_0 = \{s\}$ be the zeroth subset of Q and 0 be the corresponding state in Q' , let i be 0 , and let $last$ be 0 ;
while $i \leq last$
do begin
 for all a in Σ
 do if $\{(p, a, q) : p \in Q_i\} \neq Q_j$, for some j , $0 \leq j \leq last$
 then begin $last := last + 1$;
 $\delta' := \delta' \cup \{(i, a, last)\}$;
 $Q_{last} := \{q : p \in Q_i \text{ and } (p, a, q) \in \delta\}$
 end;
 $i := i + 1$
end {while};
 $Q' := \{i : 0 \leq i \leq last\}$;
 $F' := \{i : Q_i \cap F \neq \emptyset \text{ and } 0 \leq i \leq last\}$
end

Note that the core part of this algorithm is to take a reachable set Z of states that has not yet been examined and produce, for each symbol a in Σ , the set T_a of states that M can reach with one instruction that reads a . Formally, for all a in Σ ,

$$T_a = \{t : p \in Z \text{ and } (p, a, t) \in \delta\}.$$

There are two varieties of subset construction: **big bang** and **reachable**. With big-bang subset construction, we begin with all $2^n - 1$ subsets of the states Q , where $n = \#Q$, and, for each subset and each symbol, we determine the state set reachable with one instruction. With reachable

subset construction on the other hand, we carry out the instruction computation on only those state sets that are reachable from the state set $\{s\}$. Mathematically, these constructions are equivalent once we remove unreachable state sets after using the big-bang construction. Algorithmically, the reachable construction is more complex since we have to avoid duplication of work by considering only reachable state sets that we have not previously examined.

Thus, we have two tasks to implement efficiently: instruction computation and reachable-state-set maintenance. We focus on the task of instruction computation. In the companion paper [7], we address the issue of reachable-state-set maintenance.

3 Instruction Computation

A simple algorithm for instruction computation given a state set Z is as follows:

Algorithm Instruction Computation I
On entry: A nondeterministic finite-state machine, $M = (Q, \Sigma, \delta, s, F)$, and a state set Z .
On exit: The set $\{(Z, a, T_a) : a \in \Sigma\}$ of instructions.
begin
 for all $a \in \Sigma$
 do begin $T_a := \emptyset$;
 for all $p \in Z$
 do for all $q \in Q$
 do if $(p, a, q) \in \delta$
 then $T_a := T_a \cup \{q\}$
 end;
 return $\{(Z, a, T_a); a \in \Sigma\}$
end

It is not our aim to analyze this algorithm's execution time precisely, we just want to point out that it can perform badly.

Note that we have the Σ -loop as the outer loop to ensure that, for each a in Σ , we compute T_a completely before examining the next symbol in Σ . This algorithm takes $\Omega(\#\Sigma \cdot \#Z \cdot \#Q)$ time with a simple-minded representation and implementation. Indeed, the statement “**if** $(p, a, q) \in \delta$ ” takes $O(\#\delta)$ time if we use exhaustive search.

What execution time should be our goal? (Over the years, whenever we have discussed this issue with fellow researchers, they invariably suggested that subset construction should take time linear in the size of the output machine. Our analysis, as coarse as it is, suggests otherwise.) It seems that, at least, we have to examine each state p in Z and each instruction (p, a, q) in δ , where $a \in \Sigma$, to obtain the instruction (Z, a, T_a) . Let δ_Z denote the restriction of δ to the state set Z ; then, we should shoot for $O(\#Z + \#\delta_Z)$ time. Since $\#\delta_Z \leq \#Z \cdot \#\Sigma \cdot \#Q$, this bound is smaller than the true execution time of Algorithm I. The natural question is: *Can we do better?*

We could implement the instruction computation in a completely different way as follows:

Algorithm Instruction Computation II

On entry: A nondeterministic finite-state machine, $M = (Q, \Sigma, \delta, s, F)$, and a state set Z .

On exit: The set $\{(Z, a, T_a) : a \in \Sigma\}$ of instructions.

begin

for all $a \in \Sigma$ **do** $T_a := \emptyset$;

for all $(p, a, q) \in \delta$

do if $p \in Z$ **then** $T_a := T_a \cup \{q\}$;

return $\{(Z, a, T_a) : a \in \Sigma\}$

end

This version avoids the test of whether a specific triple (p, a, q) is in δ by using only triples that are instructions in M . The beauty of this algorithm is that it has one initialization loop that takes $O(\#\Sigma)$ time and one computation loop that takes $\Omega(\#\delta)$ time. The computation's efficiency depends on how fast we can perform the union operation, but a lower bound is $\Omega(\#\Sigma + \#\delta)$ time.

Since $\#\delta \leq \#\Sigma \cdot \#Q^2$, the run time could be excessive; however, Algorithm II suggests a more efficient implementation is possible.

In INR [4] and Grail [8], we represent δ as a sorted sequence of instructions, where we use the standard lexicographic order of tuples to sort them. In addition we represent states and symbols by nonnegative integers; hence, we represent state sets as sorted sequences of integers. Thus, in a sorted δ , all instructions with the same source state and all instructions with the same source state and the same input symbol appear as contiguous sorted subsequences. In this setting, the goal of instruction computation is to compute, for a given sorted sequence Z of integers and for all symbols $a \in \Sigma$,

the sorted sequences T_a and the corresponding instructions (Z, a, T_a) . We obtain Algorithm III as follows:

Algorithm Instruction Computation III
On entry: A nondeterministic finite-state machine, $M = (Q, \Sigma, \delta, s, F)$, and a state set Z .
On exit: The set $\{(Z, a, T_a) : a \in \Sigma\}$ of instructions.
begin
 for all $a \in \Sigma$ **do** $T_a := \emptyset$;
 for all $p \in Z$
 do for all $(p, a, q) \in p\text{-block}$
 do $T_a := T_a \cup \{q\}$;
 sort T_a ;
 return $\{(Z, a, T_a) : a \in \Sigma\}$
end

Observe that Algorithm III takes $\Omega(\#Z + \#\delta_Z)$ time; thus, we are approaching our goal. We next explore two different ways to organize this computation. The first method, Algorithm IV, is a variation of Algorithm III that makes use of an additional **instruction-block index** into the sorted δ . The idea is that the block index gives, for each source state, the location of the first instruction of its block of instructions. Thus, we can access the first instruction of an instruction block in constant time. Apart from this change, Algorithm IV is close in spirit to Algorithm III. In Grail, we do a binary search of the sorted instructions, rather than use a block index.

Algorithm Instruction Computation IV
On entry: A nondeterministic finite-state machine, $M = (Q, \Sigma, \delta, s, F)$, and a state set Z .
On exit: The set $\{(Z, a, T_a) : a \in \Sigma\}$ of instructions.
begin
 for all $a \in \Sigma$
 do begin $T_a := \emptyset$;
 for all $p \in Z$
 do while $index[p].symbol = a$
 do begin
 $T_a := T_a \cup \{index[p].target\}$;
 $index[p] := index[p] + 1$
 end
 end;
end;

```

return  $\{(Z, a, T_a) : a \in \Sigma\}$ 
end

```

The advantages of Algorithms III and IV are that they do not examine irrelevant instructions. Note that Algorithm IV processes all instructions with the same symbol at the same time. The advantage of this approach is that we need keep only one state-set variable T that we reuse for each symbol. We expect Algorithm IV to be faster than Algorithm III since it does not do any membership tests. The major problem raised by Algorithms III and IV is: How do we organize the computation of the target state set so that we obtain a sorted state set? Therefore, we still ask: *Can we do better?* We consider an alternative approach in Section 4.

4 Replacement–Selection

We propose the use of **replacement–selection**, a technique found in the standard method of implementing external sorting. In external sorting we use a k -way merge of k sorted runs, as they are called. To implement a k -way merge we use a **tournament** [5] or **heap** [1, 5, 11] of size k such that each entry in the heap is the next item in one of the runs. We use a **minimum heap** in which the root item has the smallest value in the heap and is the next item in sorted order to be appended to the output run. We **select** the root item and **replace** it with the next entry in its run. Of course, the next entry may not be the smallest item in the heap; therefore, we need to “trickle it down” to an appropriate location [1, 2, 11]. The important idea is that the heap structure is fixed; we only select and replace items. If we have k runs with a total of N items, we can k -merge them in $O(N \log k)$ time.

How is replacement–selection related to instruction computation? Basically, the idea is that if $\#Z = k$, then we treat the k instruction blocks as k runs and perform a k -way merge.

We do not give precise algorithms as we have done in the preceding section; we briefly describe the possible approaches to using replacement–selection. The most elegant technique, if not the most efficient, is that we organize a heap using the last two components of an instruction as its priority. Thus, the root of a minimum heap will hold a couple (a, q) such that $(p, a, q) \in \delta_Z$ and such that all other couples in the heap are no smaller than it (no earlier in sorted order). When we select the root value, we replace it with the couple of the next instruction in its p -block. If the

next couple has the form $(a, ?)$, it will be selected at some point during the computation of T_a . If, on the other hand, it has the form $(b, ?)$, for some $b > a$, then it will not be selected until the computation of T_a is completed. Notice that T_a will be output in sorted order as required. This algorithm takes $O(\#Z + \#\delta_{Z,a} \log \#Z)$ time to compute T_a , where $\delta_{Z,a}$ denotes the restriction of δ with Z and a .

The second approach is to treat replacement–selection conceptually and not to implement it with a heap. Essentially, we order the heap only on the symbol of the instructions; thus, destroying the sorted order of the target states. The advantage of this approach is that replacement is faster since most of the time there is no trickle down as the next instruction from a p -block will have the same symbol as the current instruction if there is nondeterminism. Of course, the target-state sequence will not be sorted, so we have to sort it in a postprocessing phase. Now, observe that we can remove the heap completely as all we are really doing is extracting all instructions with a source state from Z and an input symbol a . If we use one of the standard sorting algorithms such as Quicksort, we can compute T_a in $O(\#Z + \#\delta_{Z,a} \log \#\delta_{Z,a})$ time.

The advantage of the second approach is that we no longer have to order the couples on the fly; we have to sort the target states afterward. The second approach is faster if we can sort a set T_a fast. As we have a subset T_a of a finite set Q of states, bucket sort is applicable (and perhaps even radix sort). Thus, we can compute T_a in $O(\#Z + \#\delta_{Z,a})$ time in the expected case. Observe that this bound is the one that we were shooting for in the *worst case*. If we are able to use radix sort or bit-vector sort, then we can achieve the same time bound in the worst case. (Essentially, INR uses a bit-vector sort when the subset is large with respect to Q ; it uses a standard sorting method otherwise.) Theoretically, this is the end of the story; practically, we are not finished. Can we use radix sort or bit vector sort? Are they truly fast in practice?

Lastly, we often have to do subset construction when a machine’s program also has null-input instructions (often called ϵ or λ instructions). A null-input instruction does not read an input symbol; it has the form

$$(\text{source-state}, \lambda, \text{target-state}),$$

where λ denotes the null or empty string. We can transform a machine with null-input instructions into an equivalent machine without them using a standard algorithm; see the texts of Hopcroft and Ullman [3] and Wood [10].

This construction produces a machine that has size at most quadratic in the size of the original machine. We can avoid the explicit construction of this intermediate machine by computing its instructions on the fly as needed. Essentially, if we are computing an instruction (Z, a, T_a) , for a given state set Z and input symbol a , then, whenever there is a computation of the form

$$(p_1, \lambda, p_2) \cdots (p_{m-1}, \lambda, p_m)(p_m, a, p_{m+1}),$$

for a state p_1 in Z , then we add p_{m+1} to T_a . We can precompute the λ -reachable states from each state of the machine, so that we do not need to compute the p_{m+1} on the fly. Can we attain the goal of $O(\#Z + \#\delta_Z)$ time, in the expected and worst cases, when we also have to handle null-input instructions?

References

- [1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.
- [2] Th. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
- [3] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 2 edition, 1979.
- [4] J.H. Johnson. INR: A program for computing finite automata, 1986.
- [5] D.E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.
- [6] T. K. S. Leslie. Efficient approaches to subset construction. Technical Report CS-92-29, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1992.
- [7] T. K. S. Leslie, D. R. Raymond, and D. Wood. The expected performance of subset construction, 1996.
- [8] D. R. Raymond and D. Wood. Grail: Engineering Automata in C++, Version 2.5. Technical Report HKUST-CS96-24, Department of Computer Science, Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong, 1996.

- [9] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.
- [10] D. Wood. *Theory of Computation*. John Wiley & Sons, Inc., New York, NY, 1987.
- [11] D. Wood. *Data Structures, Algorithms, and Performance*. Addison-Wesley, Reading, MA, 1993.