Treatment of ϵ-Moves in Subset Construction

Gertjan van Noord

Alfa-informatica & BCN
University of Groningen, Netherlands
vannoord@let.rug.nl

Abstract. The paper discusses the problem of determinising finite-state automata containing large numbers of ϵ-moves. Experiments with finite-state approximations of natural language grammars often give rise to very large automata with a very large number of ϵ-moves. The paper identifies three subset construction algorithms which treat ϵ-moves. A number of experiments has been performed which indicate that the algorithms differ considerably in practice. Furthermore, the experiments suggest that the average number of ϵ-moves per state can be used to predict which algorithm is likely to perform best for a given input automaton.

1 Introduction

In experimenting with finite-state approximation techniques for context-free and more powerful grammatical formalisms (such as the techniques presented in Pereira and Wright (1997), Nederhof (1997), Evans (1997)) we have found that the resulting automata often are extremely large. Moreover, the automata contain many ϵ-moves (*jumps*). And finally, if such automata are determinised then the resulting automata are often *smaller*. It turns out that a straightforward implementation of the subset construction determinisation algorithm performs badly for such inputs.

As a motivating example, consider the definite-clause grammar that has been developed for the OVIS2 Spoken Dialogue System. This grammar is described in detail in van Noord et al. (1997). After removing the feature constraints of this grammar, and after the removal of the sub-grammar for temporal expressions, this context-free skeleton grammar was input to an implementation of the technique described in Nederhof (1997). [1] The resulting non-deterministic automaton (labelled *zovis2* below) contains 89832 states, 80935 ϵ-moves, and 80400 transitions. The determinised automaton contains only 6541 states, and 60781 transitions. Finally, the minimal automaton contains only 78 states and 526 transitions! Other grammars give rise to similar numbers. Thus, the approximation techniques yield particularly 'verbose' automata for relatively simple languages.

The experiments were performed using the FSA Utilities toolkit (van Noord, 1997). At the time, an old version of the toolkit was used, which ran into memory problems for some of these automata. For this reason, the subset construction algorithm has been re-implemented, paying special attention to the treatment of ϵ-moves. Three variants of the subset construction algorithm are identified which differ in the way ϵ-moves are treated:

per graph The most obvious and straightforward approach is sequential in the following sense. Firstly, an equivalent automaton without ϵ-moves is constructed for the input. In or-

[1] A later implementation by Nederhof (p.c.) avoids construction of the complete non-determistic automaton by determinising and minimising subautomata before they are embedded into larger subautomata.

der to do this, the transitive closure of the graph consisting of all ϵ-moves is computed. Secondly, the resulting automaton is then treated by a subset construction algorithm for ϵ-free automata.

per state For each *state* which occurs in a subset produced during subset construction, compute the states which are reachable using ϵ-moves. The results of this computation can be memorised, or computed for each state in a preprocessing step. This is the approach mentioned briefly in Johson and Wood (1997).[2]

per subset For each *subset* Q of states which arises during subset construction, compute $Q' \supseteq Q$ which extends Q with all states which are reachable from any member of Q using ϵ-moves. Such an algorithm is described in Aho, Sethi, and Ullman (1986). We extend this algorithm by memorising the ϵ-closure computation.

The motivation for this paper is the experience that the first approach turns out to be impractical for automata with very large numbers of ϵ-moves. An integration of the subset construction algorithm with the computation of ϵ-reachable states performs much better in practice. The *per subset* algorithm almost always performs better than the *per state* approach. However, for automata with a low number of jumps, the *per graph* algorithm outperforms the others.

In constructing an ϵ-free automaton the number of transitions increases. Given the fact that the input automaton already is extremely large (compared to the simplicity of the language it defines), this is an undesirable situation. An equivalent ϵ-free automaton for the example given above results in an automaton with 2353781 transitions. The implementation of *per subset* is the only variant which succeeds in determinising the input automaton of this example.

In the following section some background information concerning the *FSA Utilities* tool-box is provided. Section 3 then presents a short statement of the problem (determinise a given finite-state automaton), and a subset construction algorithm which solves this problem in the absence of ϵ-moves. Section 4 identifies three variants of the subset construction algorithm which take ϵ-moves into account. Finally, section 5 discusses some experiments in order to compare the three variants both on randomly generated automata and on automata generated by approximation algorithms.

2 FSA Utilities

The *FSA Utilities* tool-box is a collection of tools to manipulate regular expressions, finite-state automata and finite-state transducers (both string-to-string and string-to-weight transducers). Manipulations include determinisation (both for finite-state acceptors and finite-state transducers), minimisation, composition, complementation, intersection, Kleene closure, etc. Various visualisation tools are available to browse finite-state automata. The tool-box is implemented in SICStus Prolog.

The motivation for the *FSA Utilities* tool-box has been the rapidly growing interest for finite-state techniques in *computational linguistics*. The *FSA Utilities* tool-box has been developed to experiment with these techniques. The tool-box is available free of charge under Gnu General Public License.[3] The following provides an overview of the functionality of the tool-box.

[2] According to Derick Wood (p.c.), this approach has been implemented in several systems, including Howard Johnson's INR system.

[3] See `http://www.let.rug.nl/%7Evannoord/Fsa/`. The automata used in the experiments are available from the same site.

– Construction of finite automata on the basis of regular expressions. Regular expression operators include concatenation, Kleene closure, union and option (the standard regular expression operators). Furthermore the extended regular expression operators are provided: complement, difference and intersection. Symbols can be intervals of symbols, or the 'Any'-variable which matches any symbol. Regular expression operators are provided for operations on the underlying automaton, including minimisation and determinisation. Finally, we support user-defined regular expression operators.
– We also provide operators for transductions such as composition, cross-product, same-length-cross-product, domain, range, identity and inversion.
– Determinisation and Minimisation. Three different minimisation algorithms are supported: Hopcroft's algorithm (Hopcroft, 1971), Hopcroft and Ullman's algorithm (Hopcroft and Ullman, 1979), and Brzozowski's algorithm (Brzozowski, 1962).
– Determinisation and minimisation of string-to-string and string-to-weight transducers (Mohri, 1996; Mohri, 1997).
– Visualisation. Support includes built-in visualisation (Tcl/Tk, TeX+PicTeX, TeX+PsTricks, Postscript) and interfaces to third party graph visualisation software (Graphviz (dot), VCG, daVinci).
– Random generation of finite automata (an extension of the algorithm in Leslie (1995) to allow the generation of finite automata containing ϵ-moves).

3 Subset Construction

3.1 Problem statement

Let a finite-state machine M be specified by a tuple $(Q, \Sigma, \delta, S, F)$ where Q is a finite set of states, Σ is a finite alphabet, δ is a function from $Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$. Furthermore, $S \subseteq Q$ is a set of start states[4] and $F \subseteq Q$ is a set of final states.

Let ϵ-move be the relation $\{(q_i, q_j) | q_j \in \delta(q_i, \epsilon)\}$. ϵ-reachable is the reflexive and transitive closure of ϵ-move. Let ϵ-CLOSURE: $2^Q \rightarrow 2^Q$ be a function which is defined as:

$$\epsilon\text{-CLOSURE}(Q') = \{q | q' \in Q', (q', q) \in \epsilon\text{-reachable}\}$$

For any given finite-state automaton $M = (Q, \Sigma, \delta, S, F)$ there is an equivalent deterministic automaton $M' = (2^Q, \Sigma, \delta', \{Q_0\}, F')$. F' is the set of all states in 2^Q containing a final state of M, i.e., the set of subsets $\{Q_i \in 2^Q | q \in Q_i, q \in F\}$. M' has a single start state Q_0 which is the epsilon closure of the start states of M, i.e., $Q_0 = \epsilon$-CLOSURE(S). Finally,

$$\delta'(\{q_1, q_2, \ldots, q_i\}, a) = \epsilon\text{-CLOSURE}(\delta(q_1, a) \cup \delta(q_2, a) \cup \ldots \cup \delta(q_i, a))$$

An algorithm which computes M' for a given M will only need to take into account states in 2^Q which are reachable from the start state Q_0. This is the reason that for many input automata the algorithm does not need to treat all subsets of states (but note that there are automata for which all subsets are relevant, and hence exponential behaviour cannot be avoided in general).

Consider the subset construction algorithm in figure 1. The algorithm maintains a set of subsets *States*. Each subset can be either marked or unmarked (to indicate whether the subset has been treated by the algorithm); the set of unmarked subsets is sometimes referred to

[4] Note that a set of start states is required, rather than a single start state. Many operations on automata can be defined somewhat more elegantly in this way. Obviously, for deterministic automata this set should be a singleton set.

```
funct subset_construction((Q, Σ, δ, S, F))
  index_transitions(); Trans := ∅; Finals := ∅; States := ∅;
  Start =: epsilon_closure(S)
  add(Start)
  while there is an unmarked subset T ∈ States do
        mark(T)
        foreach (a, U) ∈ instructions(T) do
          U := epsilon_closure(U)
          Trans[T, a] := {U}
          add(U)
        od
  od
  return (States, Σ, Trans, {Start}, Finals)
end

proc add(U)                                   Reachable-state-set Maintenance
  if U ∉ States
    then
          add U unmarked to States
          if U ∩ F then Finals := Finals ∪ U fi
  fi
end

funct instructions(P)                         Instruction Computation
  return merge(⋃_{p∈P} transitions(p))
end

funct epsilon_closure(U)                      variant 1: No ϵ-moves
  return U
end
```

Figure 1. Subset-construction algorithm.

as the agenda. The algorithm takes such an unmarked subset T and computes all transitions leaving T. This computation is performed by the function *instructions* and is called *instruction computation* by Johson and Wood (1997).

The function *index_transitions* constructs the function *transitions* $: Q \rightarrow \Sigma \times 2^Q$. This function returns for a given state p the set of pairs (s, T) representing the transitions leaving p. Furthermore, the function *merge* takes such a set of pairs and merges all pairs with the same first element (by taking the union of the corresponding second elements). For example:

$$\mathit{merge}(\{(a,\{1,2,4\}),(b,\{2,4\}),(a,\{3,4\}),(b,\{5,6\})\}) = \{(a,\{1,2,3,4\}),(b,\{2,4,5,6\})\}$$

The procedure *add* is responsible for 'reachable-state-set maintenance', by ensuring that target subsets are added to the set of subsets if these subsets were not encountered before. Moreover, if such a new subset contains a final state, then this subset is added to the set of final states.

4 Three Variants for ϵ-Moves

The algorithm presented in the previous section does not treat ϵ-moves. In this section three possible extensions of the algorithm are identified to treat ϵ-moves.

4.1 Per graph

This variant can be seen as a straightforward implementation of the constructive proof that for any given automaton with ϵ-moves there is an equivalent one without ϵ-moves (Hopcroft and Ullman, 1979)[page 26-27].

For a given $M = (Q, \Sigma, \delta, S, F)$ this variant first computes $M' = (Q, \Sigma, \delta', S', F)$, where $S' = \epsilon$-CLOSURE(S), and $\delta'(q, a) = \epsilon$-CLOSURE$(\delta(q, a))$. The function ϵ-CLOSURE is computed by using a standard transitive closure algorithm for directed graphs: this algorithm is applied to the directed graph consisting of all ϵ-moves of M. Such an algorithm can be found in several textbooks (see, for instance, Cormen, Leiserson, and Rivest (1990)).

The advantage of this approach is that the subset construction algorithm does not need to be modified at all. Moreover, the transitive closure algorithm is fired only once (for the full graph), whereas the following two variants call a specialised transitive closure algorithm possibly many times.

4.2 Per subset and per state

The *per subset* and the *per state* algorithm use a variant of the transitive closure algorithm for graphs. Instead of computing the transitive closure of a given graph, this algorithm only computes the closure for a given set of states. Such an algorithm is given in figure2.

funct *closure*(T)
    $D =: \emptyset$
    foreach $t \in T$ **do** add t unmarked to D **od**
    while there is an unmarked state $t \in D$ **do**
            mark(t)
            foreach $q \in \delta(t, \epsilon)$ **do**
                if $q \notin D$ **then** add q unmarked to D **fi**
            od
    od
    return D
end

Figure 2. Epsilon-closure Algorithm

In either of the two integrated approaches, the subset construction algorithm is initialised with an agenda containing a single subset which is the ϵ-CLOSURE of the set of start-states of the input; furthermore, the way in which new transitions are computed also takes the effect of ϵ-moves into account. Both differences are accounted for by an alternative definition of the *epsilon_closure* function.

The approach in which the transitive closure is computed for one state at a time is defined by the following definition of the *epsilon_closure* function. Note that we make sure that the transitive closure computation is only performed once for each input state, by memorising the *closure* function.

funct $\mathit{epsilon\_closure}(U)$ — variant 2: per state
  return $\bigcup_{u \in U} \mathit{memo}(\mathit{closure}(\{u\}))$
end

In the case of the *per subset* approach the closure algorithm is applied to each subset. We also memorise the closure function, in order to ensure that the closure computation is performed only once for each subset. This can be useful since the same subset can be generated many times during subset construction. The definition simply is:

funct $\mathit{epsilon\_closure}(U)$ — variant 3: per subset
  return $\mathit{memo}(\mathit{closure}(U))$
end

The motivation for *per state* approach may be the insight that in this case the closure algorithm is called at most $|Q|$ times. In contrast, in the *per subset* approach the transitive closure algorithm may need to be called $2^{|Q|}$ times. On the other hand, in the *per state* approach some overhead must be accepted for computing the union of the results for each state. Moreover, in practice the number of subsets is often much smaller than $2^{|Q|}$. In some cases, the number of reachable subsets is smaller than the number of states encountered in those subsets.

5 Experiments

Two sets of experiments have been performed. In the first set of experiments a number of random automata is generated according to a number of criteria (based on Leslie (1995)) . In the second set of experiments, results are provided for a number of (much larger) automata that surfaced during actual development work on finite-state approximation techniques.

Random automata. Firstly, consider a number of experiments for randomly generated automata. Following Leslie (1995), the *absolute transition density* of an automaton is defined as the number of transitions divided by the square of the number of states times the number of symbols (i.e. the number of transitions divided by the number of possible transitions). *Deterministic transition density* is the number of transitions divided by the number of states times the number of symbols (i.e. the ratio of the number of transitions and the number of possible transitions in a deterministic machine). Leslie (1995) shows that *deterministic transition density* is a reliable measure for the difficulty of subset construction. Exponential blow-up can be expected for input automata with deterministic transition density of around 2.[5]

A number of automata were generated randomly, according to the number of states, symbols, and transition density. The random generator makes sure that all states are reachable from the start state. For the first experiment, a number of automata was randomly generated, consisting of 15 symbols, and 15, 20, 25, 100 or 1000 states, using various densities (and no ϵ-moves).

[5] Leslie uses the terms *absolute density* and *deterministic density*.

The results are summarised in figure 3. Only a single result is given since each of the implementations works equally well in the absence of ϵ-moves.[6]

A new concept called *absolute jump density* is introduced to specify the number of ϵ-moves. It is defined as the number of ϵ-moves divided by the square of the number of states (i.e., the probability that an ϵ-move exists for a given pair of states). Furthermore, *deterministic jump density* is the number of ϵ-moves divided by the number of states (i.e., the average number of ϵ-moves which leave a given state). In order to measure the differences between the three implementations, a number of automata has been generated consisting of 15 states and 15 symbols, using various transition densities between 0.01 and 0.3 (for larger densities the automata tend to collapse to an automaton for Σ^*). For each of these transition densities, jump densities were chosen in the range 0.01 to 0.24 (again, for larger values the automaton collapses). In figure 4 the outcomes of this experiment are summarised by listing the average amount of CPU-time required per deterministic jump density (for each of the three algorithms). Thus, every dot represents the average for determinising a number of different input automata with various absolute transition densities and the same deterministic jump density. The figures 5, 6 and 7 summarise similar experiments using input automata with 20, 25 and 100 states.[7]

The striking aspect of these experiments is that the *per graph* algorithm is more efficient for lower deterministic jump densities, whereas, if the deterministic jump density gets larger, the *per subset* algorithm is more efficient. The turning point is around a deterministic jump density between 1 and 1.5, where it seems that for larger automata the turning point occurs at a lower determinisic jump density. Interestingly, this generalisation is supported by the experiments on automata which were generated by approximation techniques (although the results for randomly generated automata are more consistent than the results for 'real' examples).

Experiment: Automata generated by approximation algorithms The automata used in the previous experiments were randomly generated, according to a number of criteria. However, it may well be that in practice the automata that are to be treated by the algorithm have typical properties which were not reflected in this test data. For this reason results are presented for a number of automata that were generated using approximation techniques for context-free grammars (Pereira and Wright, 1997; Nederhof, 1997; Evans, 1997). In particular, a number of automata has been used generated by Mark-Jan Nederhof using the technique described in Nederhof (1997). In addition, a small number of automata have been used which were generated using the technique of Pereira and Wright (1997) (as implemented by Nederhof).

The automata typically contain lots of jumps. Moreover, the number of states of the resulting automaton is often *smaller* than the number of states in the input automaton. Results are given in table 1. One of the most striking examples is the *ygrim* automaton consisting of 3382 states

[6] CPU-time was measured on a HP 9000/780 machine running HP-UX 10.20, 240Mb, with SICStus Prolog 3 #3. For comparison with an "industrial strength" implementation, we have applied the determiniser of AT&T's FSM utilities for the same examples. The results show that for automata with very small transition densities FSM is faster (up to 2 or 3 times as fast), but for automata with larger densities the results are very similar, in some cases our Prolog implementation is even faster. Note finally that our timings do include IO, but not the start-up of the Prolog engine.

[7] We also provide the results for FSM again; we used the pipe `fsmrmepsilon | fsmdeterminize` . According to Fernando Pereira (pc) the comparison is less meaningful in this case because the fsmrmepsilon program treats weighted automata. This generalisation requires some overhead also in case no weights are used (for the determiniser this generalisation does not lead to any significant overhead). Pereira mentions furthermore that FSM used to include a determiniser with integrated treatment of jumps. Because this version could not (easily) be generalised for weighted automata it was dropped from the tool-set.

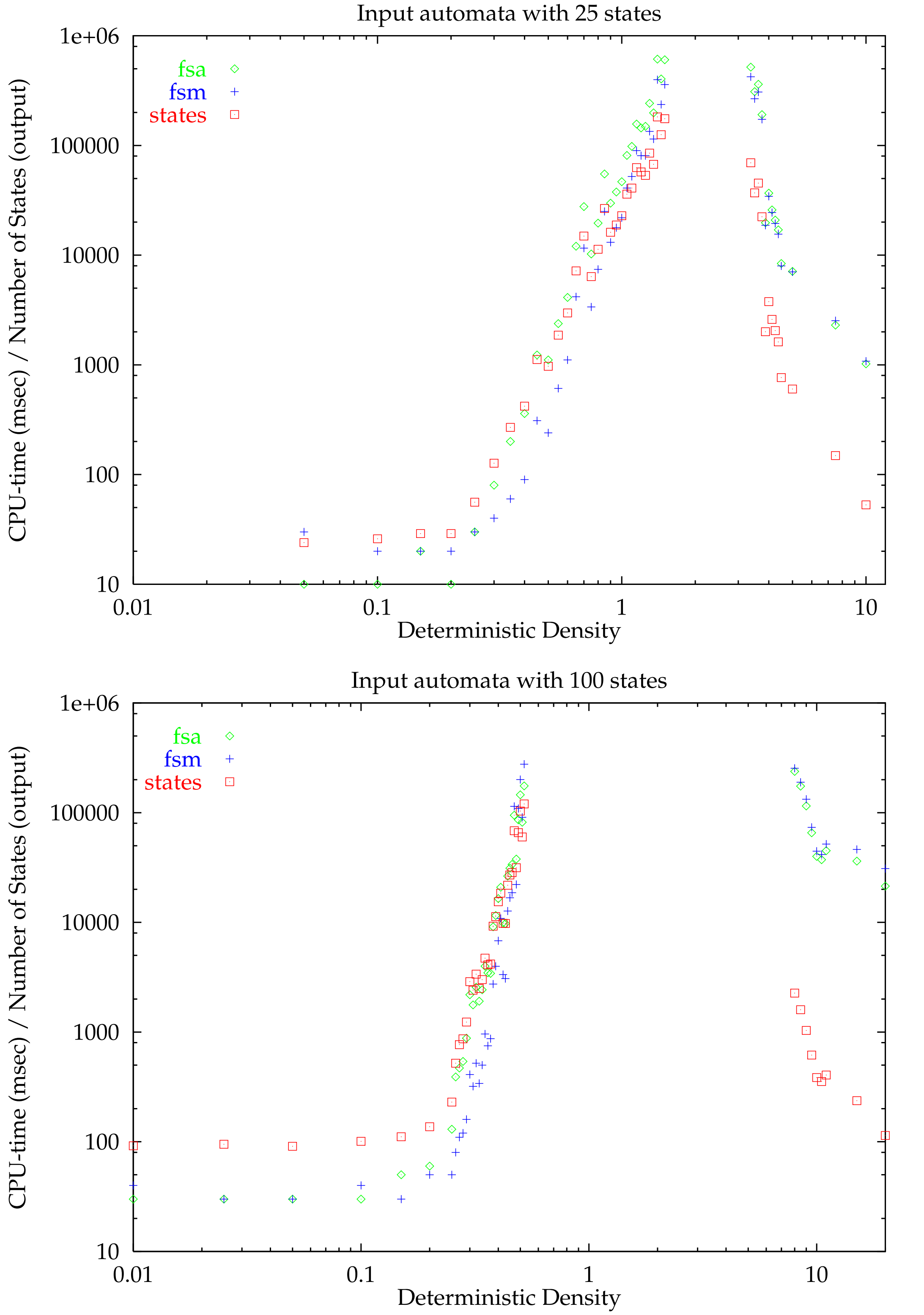

Figure 3. Deterministic transition density versus CPU-time in msec. The input automata have no ϵ-moves.

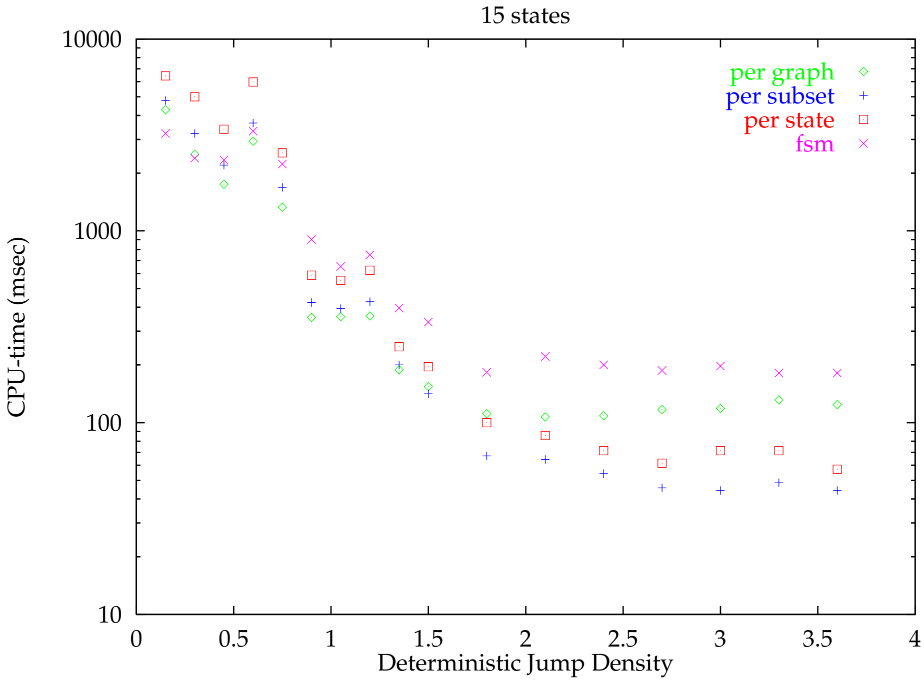

Figure 4. Average amount of CPU-time versus jump density for each of the three algorithms, and FSM. Input automata have 15 states. Absolute transition densities: 0.01-0.3.

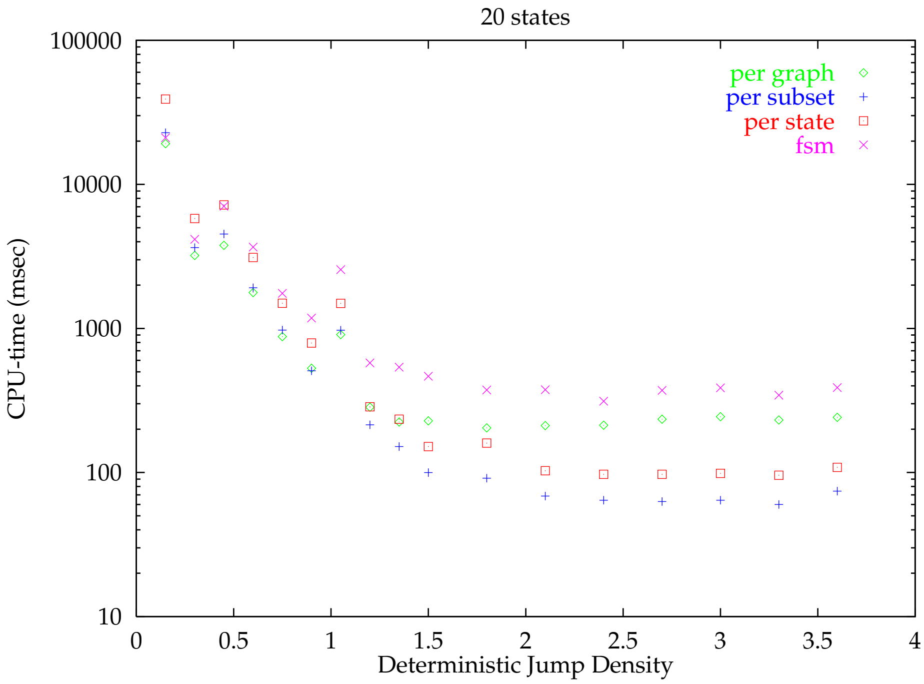

Figure 5. Average amount of CPU-time versus jump density for each of the three algorithms, and FSM. Input automata have 20 states. Absolute transition densities: 0.01-0.3.

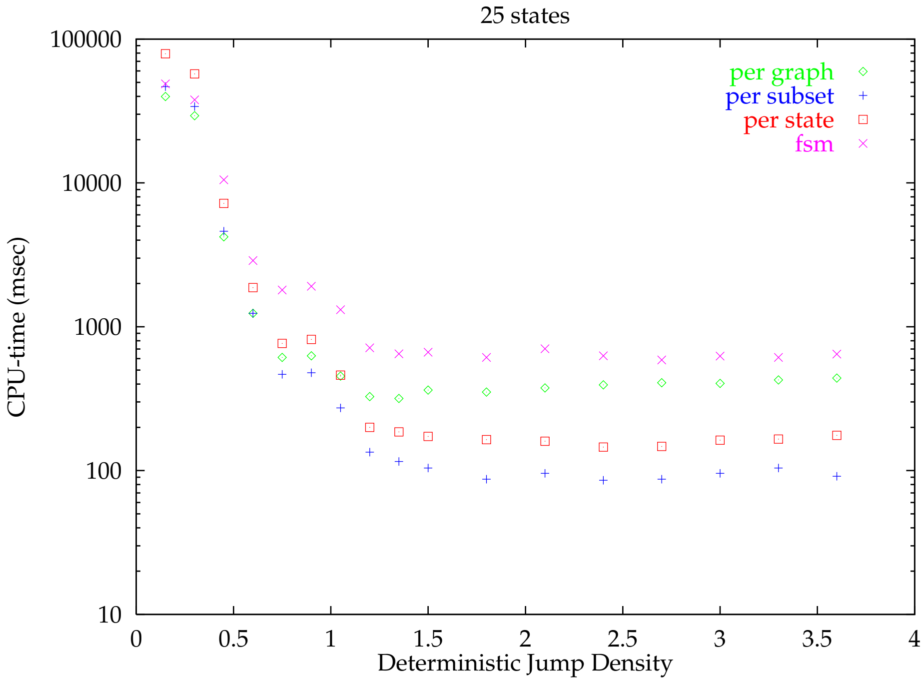

Figure 6. Average amount of CPU-time versus deterministic jump density for each of the three algorithms, and FSM. Input automata have 25 states. Absolute transition densities: 0.01-0.3.

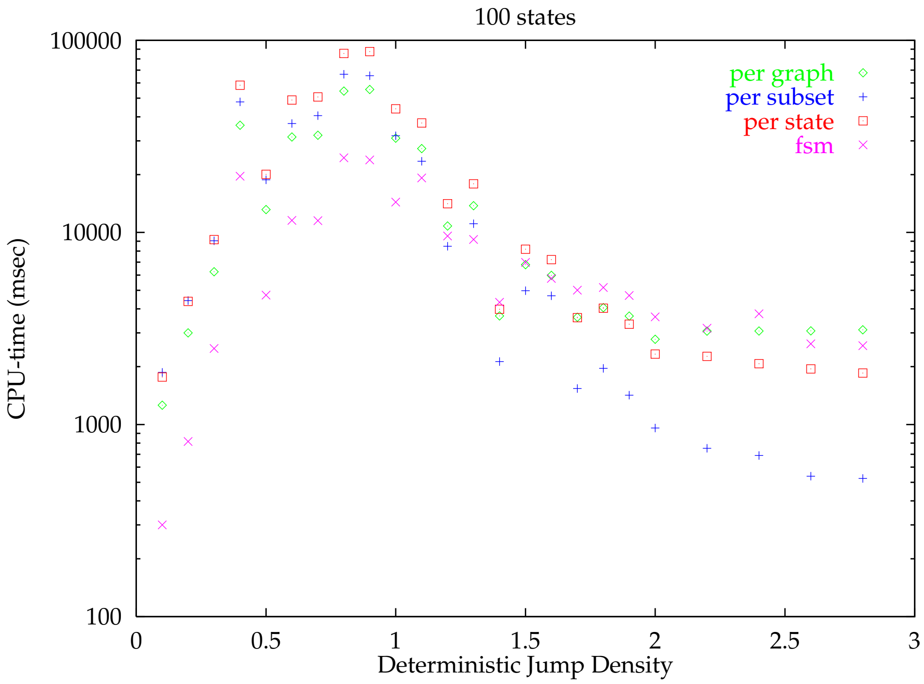

Figure 7. Average amount of CPU-time versus deterministic jump density for each of the three algorithms, and FSM. Input automata have 100 states. Absolute transition densities: 0.001-0.0035.

and 10571 jumps. For this example, the *per graph* implementation ran out of memory (after a long time), whereas the *per subset* algorithm produced the determinised automaton relatively quickly. The FSM implementation took much longer for this example (whereas for many of the other examples it performs better than our implementations). Note that this example has the highest number of jumps per number of states ratio.

	input automaton			CPU-time (msec)			
Id	#states	# transitions	# jumps	graph	subset	state	*FSM*
griml.n	238	43	485	2060	100	140	40
g9a	342	58	478	260	70	70	30
g7	362	424	277	180	240	200	60
g15	409	90	627	280	130	180	40
ovis5.n	417	702	130	290	320	380	190
g9	438	313	472	560	850	640	110
g11	822	78	1578	1280	160	160	60
g8	956	2415	330	500	500	610	140
g14	1048	403	1404	1080	1240	730	120
ovis4.n	1424	2210	660	2260	2220	2870	1310
g13	1441	1006	1404	2400	3780	2550	440
rene2	1800	2597	96	440	530	600	200
ovis9.p	1868	2791	3120	83340	80400	87040	52560
ygrim	3382	5422	10571	-	2710	70140	784910
ygrim.p	48062	63704	122095	-	1438960	-	8575850
java19	54369	28333	59394	130130	55290	64420	8470
java16	64210	43935	43505	67180	24200	31770	6370
zovis3	88156	78895	79437	-	968160	-	768440
zovis2	89832	80400	80935	-	1176650	-	938040

Table 1. Results for automata generated by approximation algorithms. The dashes in the table indicate that the corresponding algorithm ran out of memory (after a long period of time) for that particular example.

6 Conclusion

We have discussed three variants of the subset-construction algorithm for determinising finite automata. The experiments support the following conclusions:

- the *per graph* variant works best for automata with a limited number of jumps
- the *per subset* variant works best for automata with a large number of jumps
- the *per state* variant almost never outperforms both of the two other variants
- typically, if the deterministic jump density of the input is less than 1, then the *per graph* variant outperforms the *per subset* variant. If this value is larger than 1.5, then *per subset* outperforms *per graph*.
- the *per subset* approach is especially useful for automata generated by finite-state approximation techniques, because those techniques often yield automata with very large number of ϵ-moves.

Acknowledgements

I am grateful to Mark-Jan Nederhof for support, and for providing me with lots of (often dreadful) automata generated by his finite-state approximation tools.

References

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers. Principles, Techniques and Tools*. Addison Wesley.

Brzozowski, J.A. 1962. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical theory of Automata*. Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y., pages 529–561. Volume 12 of MRI Symposia Series.

Cormen, Leiserson, and Rivest. 1990. *Introduction to Algorithms*. Cambridge Mass.: MIT Press.

Evans, Edmund Grimley. 1997. Approximating context-free grammars with a finite-state calculus. In *35th Annual Meeting of the Association for Computational Linguistics and 8th Conference of the European Chapter of the Association for Computational Linguistics*, pages 452–459, Madrid.

Hopcroft, John E. 1971. An n log n algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*. Academic Press, pages 189–196.

Hopcroft, John E. and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley.

Johson, J. Howard and Derick Wood. 1997. Instruction computation in subset construction. In Darrell Raymond, Derick Wood, and Sheng Yu, editors, *Automata Implementation*. Springer Verlag, pages 64–71. Lecture Notes in Computer Science 1260.

Leslie, Ted. 1995. Efficient approaches to subset construction. Master's thesis, Computer Science, University of Waterloo.

Mohri, Mehryar. 1996. On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering*, 2:61–80. Originally appeared in 1994 as Technical Report, institut Gaspard Monge, Paris.

Mohri, Mehryar. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311.

Nederhof, M.J. 1997. Regular approximations of CFLs: A grammatical view. In *International Workshop on Parsing Technologies*, Massachusetts Institute of Technology, September.

van Noord, Gertjan. 1997. FSA Utilities: A toolbox to manipulate finite-state automata. In Darrell Raymond, Derick Wood, and Sheng Yu, editors, *Automata Implementation*. Springer Verlag. Lecture Notes in Computer Science 1260.

van Noord, Gertjan, Gosse Bouma, Rob Koeling, and Mark-Jan Nederhof. 1997. Robust grammatical analysis for spoken dialogue systems. Article submitted to Journal of Natural Language Engineering. Draft availabel from `http://www.let.rug.nl/~vannoord/`.

Pereira, Fernando C.N. and Rebecca N. Wright. 1997. Finite-state approximation of phrase-structure grammars. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*. MIT Press, Cambridge, pages 149–173.