Treatment of Epsilon Moves in Subset Construction

Gertjan van Noord*
Rijksuniversiteit Groningen

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 and compares a number of subset construction algorithms which treat ϵ-moves. Experiments have been performed which indicate that the algorithms differ considerably in practice, both with respect to the size of the resulting deterministic automaton, and with respect to practical efficiency. Furthermore, the experiments suggest that the average number of ϵ-moves per state can be used to predict which algorithm is likely to be the fastest for a given input automaton.

1 Introduction

1.1 Finite-state Language Processing

An important problem in computational linguistics is posed by the fact that the grammars which are typically hypothesised by linguists are unattractive from the point of view of computation. For instance, the number of steps required to analyse a sentence of n words is n^3 for context-free grammars. For certain linguistically more attractive grammatical formalisms it can be shown that no upper-bound to the number of steps required to find an analysis can be given. The human language user, however, seems to process in linear time; humans understand longer sentences with no noticeable delay. This implies that neither context-free grammars nor more powerful grammatical formalisms are likely models for human language processing. An important issue therefore is how the linearity of processing by humans can be accounted for.

A potential solution to this problem concerns the possibility of *approximating* an underlying general and abstract grammar by techniques of a much simpler sort. The idea that a competence grammar might be approximated by finite-state means goes back to early work by Chomsky (Chomsky, 1963; Chomsky, 1964). There are essentially three observations which motivate the view that the processing of natural language is finite-state:

1. humans have a finite (small, limited, fixed) amount of memory available for language processing
2. humans have problems with certain grammatical constructions, such as center-embedding, which are impossible to describe by finite-state means (Miller and Chomsky, 1963)
3. humans process natural language very efficiently (in linear time)

---

* Alfa-informatica & BCN. E-mail: vannoord@let.rug.nl

1.2 Finite-state Approximation and ϵ-moves

In experimenting with finite-state approximation techniques for context-free and more powerful grammatical formalisms (such as the techniques presented in Black (1989), Pereira and Wright (1991), Rood (1996), Pereira and Wright (1997), Evans (1997), Nederhof (1997), Nederhof (1998), Johnson (1998)) 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. In this paper we consider a number of variants of the subset-construction algorithm which differ in their treatment of ϵ-moves.

Although we have observed that finite-state approximation techniques typically yield automata with large amounts of ϵ-moves, this is obviously not a necessity. Instead of trying to improve upon determinisation techniques for such automata it might be more fruitful, perhaps, to try to improve these approximation techniques in such a way that more compact automata are produced. [1] However, because research into finite-state approximation is still of an exploratory and experimental nature, it can be argued that more robust determinisation algorithms do still have a role to play: it can be expected that approximation techniques are much easier to define and implement if the resulting automaton is allowed to be non-deterministic and to contain ϵ-moves.

Note furthermore that even if our primary motivation is in finite-state approximation, the problem of determinising finite-state automata with ϵ-moves may be relevant in other areas of language research as well.

1.3 Subset construction and ϵ-moves

The experiments were performed using the *FSA Utilities*. The *FSA Utilities* tool-box (van Noord, 1997; van Noord, 1999; Gerdemann and van Noord, 1999; van Noord and Gerdemann, 1999) is a collection of tools to manipulate regular expressions, finite-state automata and finite-state transducers. Manipulations include determinisation, 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, and is available free of charge under Gnu General Public License via anonymous ftp at ftp://ftp.let.rug.nl/pub/vannoord/Fsa/, and via the web at http://www.let.rug.nl/˜vannoord/Fsa/. At the time of our initial experiments with finite-state approximation, an old version of the tool-box 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 order 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. Different variants of *per graph* can be identified, depending on the implementation of the ϵ-removal step.

per state For each *state* which occurs in a subset produced during subset construc-

---

[1] Indeed, a later implementation by Nederhof avoids construction of the complete non-determistic automaton by minimising sub-automata before they are embedded into larger sub-automata.

tion, 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 Johnson 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).

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 for such automata.

Section 2 presents a short statement of the problem (how to determinise a given finite-state automaton), and a subset construction algorithm which solves this problem in the absence of ϵ-moves. Section 3 defines a number of subset construction algorithms which differ with respect to the treatment of ϵ-moves. Most aspects of the algorithms are not new and have been described elsewhere, and/or were incorporated in previous implementations; a comparison of the different algorithms had not been performed previously. We provide a comparison with respect to the size of the resulting deterministic automaton (in section 3) and practical efficiency (in section 4). Section 4 provides experimental results both for randomly generated automata and for automata generated by approximation algorithms. Our implementations of the various algorithms are also compared with AT&T's FSM utilities (Mohri, Pereira, and Riley, 1998), to establish that the experimental differences we find between the algorithms are truly caused by differences in the algorithm (as opposed to accidental implementation details).

2 Subset Construction

2.1 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 and $F \subseteq Q$ is a set of final states. [3]

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:

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

Furthermore, we write ϵ-CLOSURE$^{-1}(Q')$ for the set $\{q | q' \in Q', (q, q') \in \epsilon\text{-reachable}\}$.

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,

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

---

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

[3] 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 (including *per graph^t* discussed below). Obviously, for deterministic automata this set should be a singleton set.

```
funct subset_construction($(Q, \Sigma, \delta, S, F)$)
  index_transitions(); Trans $:= \emptyset$; Finals $:= \emptyset$; States $:= \emptyset$;
  Start $:=$ epsilon_closure($S$)
  add($Start$)
  while there is an unmarked subset $T \in$ States do
        mark($T$)
        foreach $(a, U) \in$ instructions($T$) do
          $U :=$ epsilon_closure($U$)
          Trans$[T, a] := \{U\}$
          add($U$)
        od
  od
  return (States, $\Sigma$, Trans, $\{$Start$\}$, Finals)
end

proc add($U$)                          Reachable-state-set Maintenance
  if $U \notin$ States
    then add $U$ unmarked to States
         if $U \cap F$ then Finals $:=$ Finals $\cup \{U\}$ fi
  fi
end

funct instructions($P$)                       Instruction Computation
  return merge($\bigcup_{p \in P}$ transitions($p$))
end

funct epsilon_closure($U$)                     variant 1: No $\epsilon$-moves
  return $U$
end
```

Figure 1
Subset-construction algorithm.

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 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 Johnson and Wood (1997).

The function *index_transitions* constructs the function *transitions*: $Q \rightarrow \Sigma \times 2^Q$ which 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.

3 Variants for ϵ-Moves

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

3.1 Per graph

In the *per graph* variant two steps can be identified. In the first step, *efree*, an equivalent ϵ-free automaton is constructed. In the second step this ϵ-free automaton is determinised using the subset construction algorithm. The advantage of this approach is that the subset construction algorithm can remain simple because the input automaton is ϵ-free.

An algorithm for *efree* is described for instance in Hopcroft and Ullman (1979)[page 26-27]. The main ingredient of *efree* is the construction of the function ϵ-CLOSURE, which can be 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)).

For a given finite-state automaton $M = (Q, \Sigma, \delta, S, F)$ *efree* computes $M' = (Q, \Sigma, \delta', S', F')$, where $S' = \epsilon\text{-CLOSURE}(S)$, $F' = \epsilon\text{-CLOSURE}^{-1}(F)$, and $\delta'(p, a) = \{q | q' \in \delta(p', a), p' \in \epsilon\text{-CLOSURE}^{-1}(p), q \in \epsilon\text{-CLOSURE}(q')\}$. Instead of using ϵ-CLOSURE on both the source and target side of a transition, *efree* can be optimised in two different ways by using ϵ-CLOSURE only on one side:

- *efree^t*: $M' = (Q, \Sigma, \delta', S', F)$, where $S' = \epsilon\text{-CLOSURE}(S)$, and $\delta'(p, a) = \{q | q' \in \delta(p, a), q \in \epsilon\text{-CLOSURE}(q')\}$.
- *efree^s*: $M' = (Q, \Sigma, \delta', S, F')$, where $F' = \epsilon\text{-CLOSURE}^{-1}(F)$, and $\delta'(p, a) = \{q | q \in \delta(p', a), p' \in \epsilon\text{-CLOSURE}^{-1}(p)\}$.

Although both variants appear very similar, there are some differences. Firstly, *efree^t* might introduce states which are not *co-accessible*: states from which no path exists to a final state; in contrast, *efree^s* might introduce states which are not *accessible*: states from which no path exists from the start state. A straightforward modification of both algorithms is possible to ensure that these states are not present in the output. Thus *efree^{t,c}* ensures that all states in the resulting automaton are co-accessible; *efree^{s,a}* ensures that all states in the resulting automaton are *accessible*. As a consequence, the size of the determinised machine is in general smaller if *efree^{t,c}* is employed, because states which were not co-accessible (in the input) are removed (this is therefore an additional benefit of *efree^{t,c}*; the fact that *efree^{s,a}* removes accessible states has no effect on the size of the determinised machine because the subset construction algorithm already ensures accessibility anyway).

Secondly, it turns out that applying *efree^t* in combination with the subset-construction algorithm generally produces smaller automata than *efree^s* (even if we ignore the benefit of ensuring co-accessibility). An example is presented in figure 2. The differences can be quite significant. This is illustrated in figure 3.

Below we will write *per graph^X* to indicate the non-integrated algorithm based on *efree^X*.

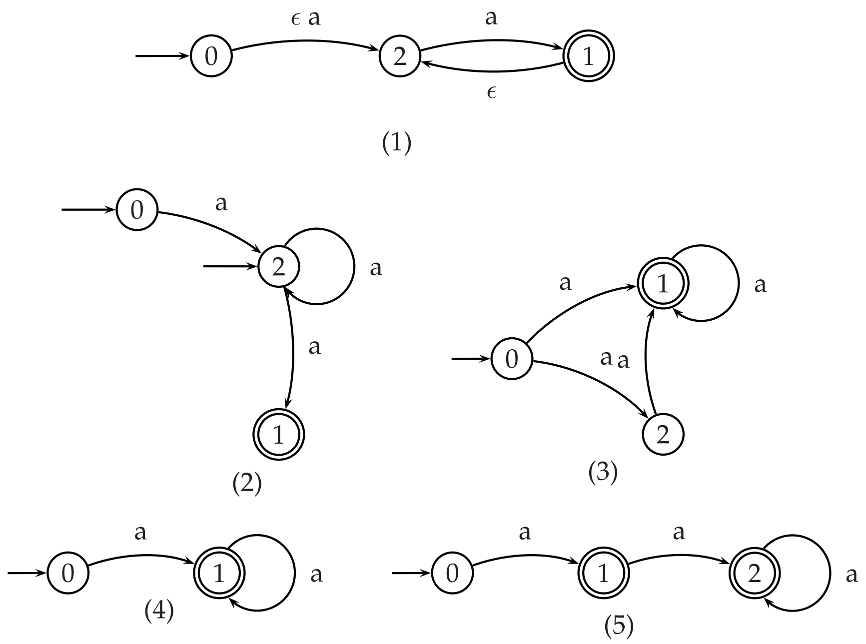

Figure 2
Illustration of the difference in size between two variants of *efree*. (1) is the input automaton. The result of *efree*^{t} is given in (2); (3) is the result of *efree*^{s}. (4) and (5) are the result of applying the subset construction to the result of *efree*^{t} and *efree*^{s}, respectively.

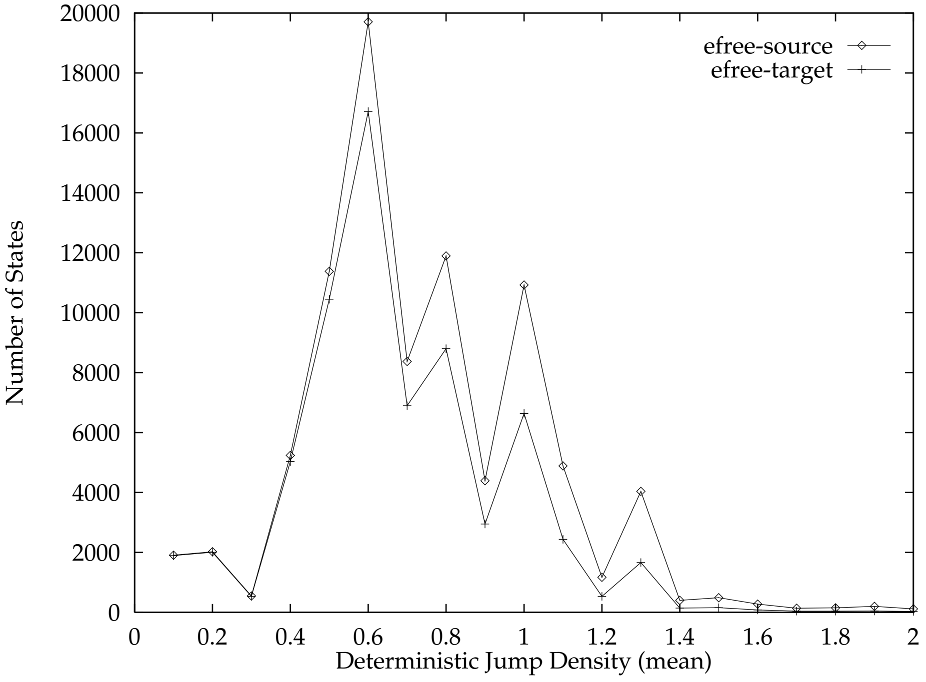

Figure 3
Difference in sizes of deterministic automata constructed with either *efree*^{s} or *efree*^{t}, for randomly generated input automata consisting of 100 states, 15 symbols, and various numbers of transitions and jumps (cf. section 4). Note that all states in the input are co-accessible; the difference in size is due solely to the effect illustrated in figure 2.

```
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 4
Epsilon-closure Algorithm

3.2 Per subset and per state

Next we discuss two variants (*per subset* and *per state*) in which the treatment of ϵ-moves is integrated with the subset construction algorithm. We will show later that such an integrated approach is in practice often more efficient than the *per graph* approach if there are many ϵ-moves. The *per subset* and *per state* approaches are also more suitable for a *lazy* implementation of the subset construction algorithm (in such a lazy implementation subsets are only computed with respect to a given input string).

The *per subset* and the *per state* algorithms use a simplified 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 figure 4.

In both 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; the full computation is memorised as well. [4]

```
funct epsilon_closure($U$)                                variant 2: per state
  return memo($\bigcup_{u \in U}$ memo(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 epsilon_closure($U$)                                variant 3: per subset
```

---

[4] This is an improvement over the algorithm given in a preliminary version of this paper (van Noord, 1998).

```
    return memo(closure(U))
end
```

The motivation for the *per state* variant is 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.

3.3 Implementation

In order to implement the algorithms efficiently in Prolog, it is important to use efficient data-structures. In particular, we use an implementation of (non-updatable) arrays based on the N+K trees of O'Keefe (1990, pp.142-145) with N=95 and K=32. On top of this datastructure, a hash array is implemented using the SICStus library predicate `term_hash/4` which constructs a key for a given term. In such hashes, a value in the underlying array is a partial list of key-value pairs; thus collisions are resolved by chaining. This provides efficient access in practice, although such arrays are quite memory-intensive: care must be taken to ensure that the deterministic algorithms indeed are implemented without introducing choice-points during runtime.

4 Experiments

Two sets of experiments have been performed. In the first set of experiments, random automata are 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. [5]

Random automata. Firstly, we report on 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 multiplied by the number of symbols (i.e. the number of transitions divided by the maximum number of 'possible' transitions, or, in other words, the probability that a possible transition in fact exists). *Deterministic transition density* is the number of transitions divided by the number of states multiplied by the number of symbols (i.e. the ratio of the number of transitions and the maximum number of 'possible' transitions in a deterministic machine).

In both of these definitions, the number of transitions should be understood as the the number of non-duplicate transitions which do not lead to a sink state. A sink state is a state from which there exists no sequence of transitions to a final state. In the randomly generated automata, states are accessible and co-accessible by construction; sink states and associated transitions are not represented.

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.[6] He concludes (page 66):

> [. . .] randomly generated automata exhibit the maximum execu-

---

[5] All the automata used in the experiments are freely available from http://www.let.rug.nl/~vannoord/Fsa/.

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

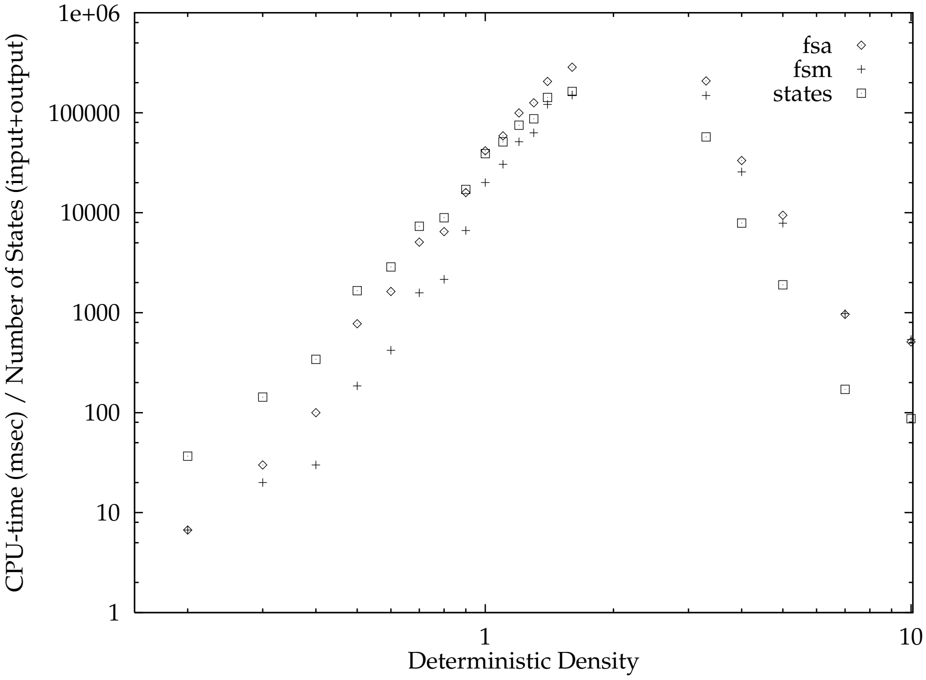

Figure 5
Deterministic transition density versus CPU-time in msec. The input automata have 25 states, 15 symbols, and no ϵ-moves. *fsa* represents the CPU-time required by our FSA6 implementation; *fsm* represents the CPU-time required by AT&T's FSM library; *states* represents the sum of the number of states of the input and output automata.

> tion time, and the maximum number of states, at an approximate deterministic density of 2. Most of the area under the curve occurs within 0.5 and 2.5 deterministic density—this is the area in which subset construction is expensive.
>
> **Conjecture.** For a given NFA, we can compute the expected numbers of states and transitions in the corresponding DFA, produced by subset construction, from the deterministic density of the NFA. In addition, this functional relationship gives rise to a Poisson-like curve with its peak approximately at a deterministic density of 2.

A number of automata were generated randomly, according to the number of states, symbols, and transition density. For the first experiment, automata were generated consisting of 15 symbols, 25 states, and various densities (and no ϵ-moves). The results are summarised in figure 5. CPU-time was measured on a HP 9000/785 machine running HP-UX 10.20. Note that our timings do not include the start-up of the Prolog engine, nor the time required for garbage collection.

In order to establish that the differences we obtain later are genuinely due to differences in the underlying algorithm, and not due to 'accidental' implementation details, we have compared our implementation with the determiniser of AT&T's FSM utilities (Mohri, Pereira, and Riley, 1998). For automata without ϵ-moves we establish that FSM normally is faster: for automata with very small transition densities FSM is up to four times as fast, for automata with larger densities the results are similar.

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, deterministic jump densities were chosen in the range 0 to 2.5 (again, for larger values the automata tend to collapse). In figures 6 to 9 the outcomes of these experiments are summarised by listing the average amount of CPU-time required per deterministic jump density (for each of the algorithms), using automata with 15, 20, 25 and 100 states respectively. 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 striking aspect of these experiments is that the integrated *per subset* and *per state* variants are much more efficient for larger deterministic jump density. The *per graph*^{t} is typically the fastest algorithm of the non-integrated versions. However, in these experiments all states in the input are co-accessible by construction; and moreover, all states in the input are final states. Therefore, the advantages of the *per graph*^{t,c} algorithm could not be observed here.

The turning point is around a deterministic jump density of around 0.8: for smaller densities the *per graph*^{t} is typically slightly faster; for larger densities the *per state* algorithm is much faster. For densities beyond 1.5, the *per subset* algorithm tends to perform better than the *per state* algorithm. 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).

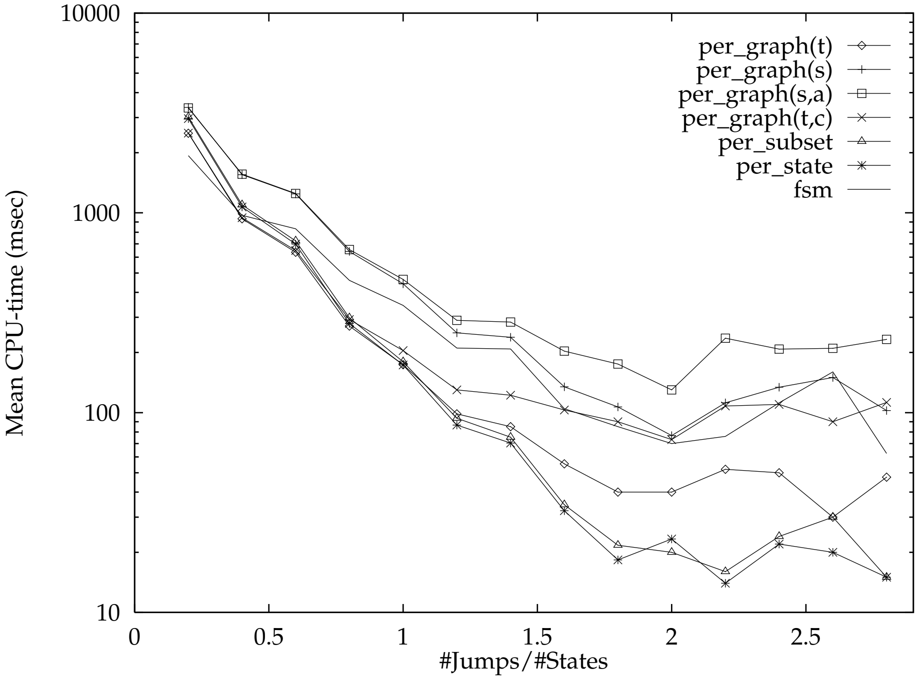

Figure 6
Average amount of CPU-time versus jump density for each of the algorithms, and FSM. Input automata have 15 states. Absolute transition densities: 0.01-0.3.

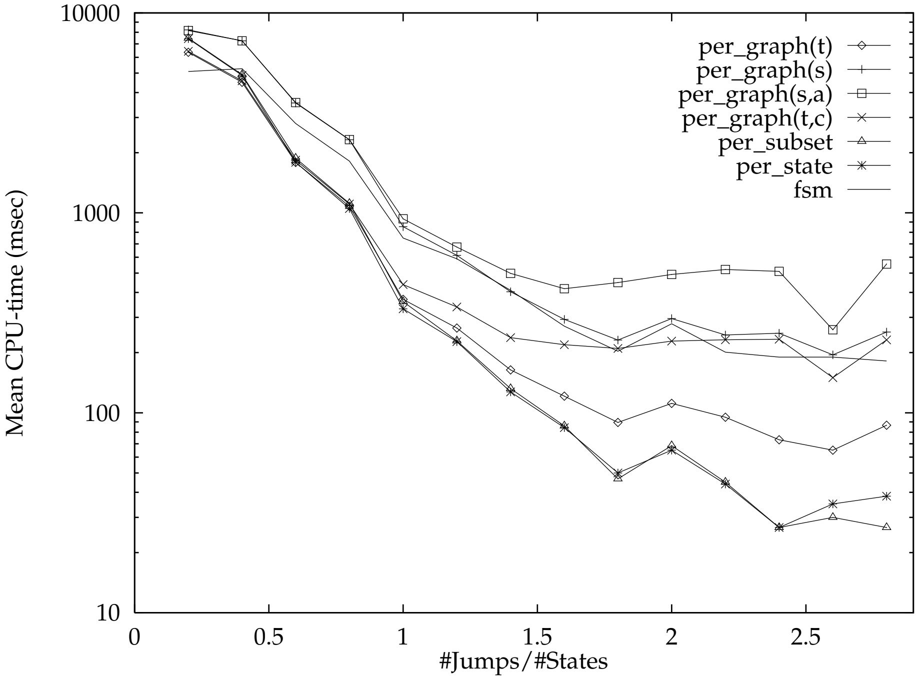

Figure 7
Average amount of CPU-time versus jump density for each of the algorithms, and FSM. Input automata have 20 states. Absolute transition densities: 0.01-0.3.

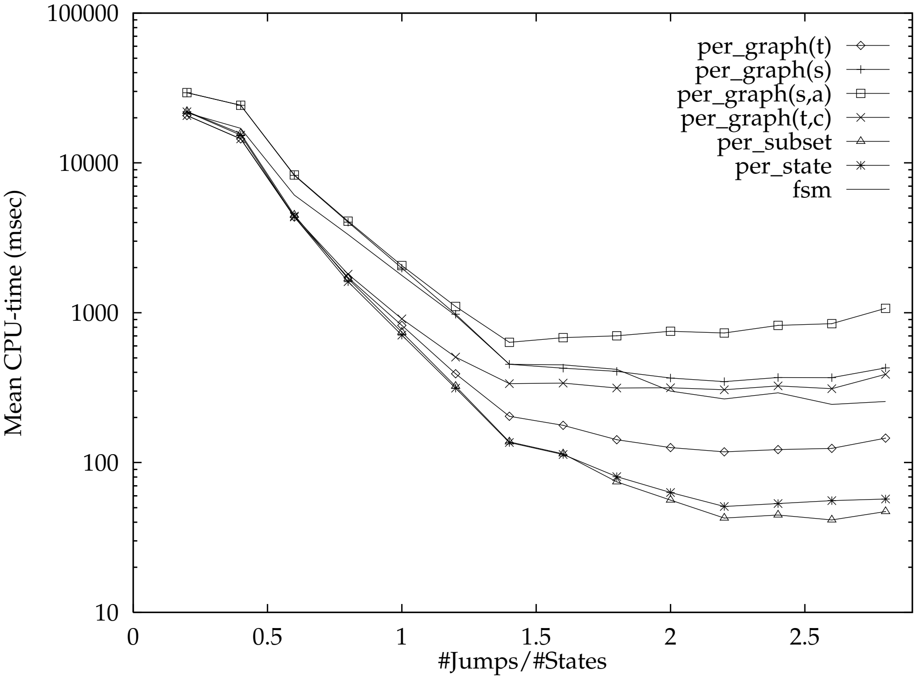

Figure 8
Average amount of CPU-time versus deterministic jump density for each of the algorithms, and FSM. Input automata have 25 states. Absolute transition densities: 0.01-0.3.

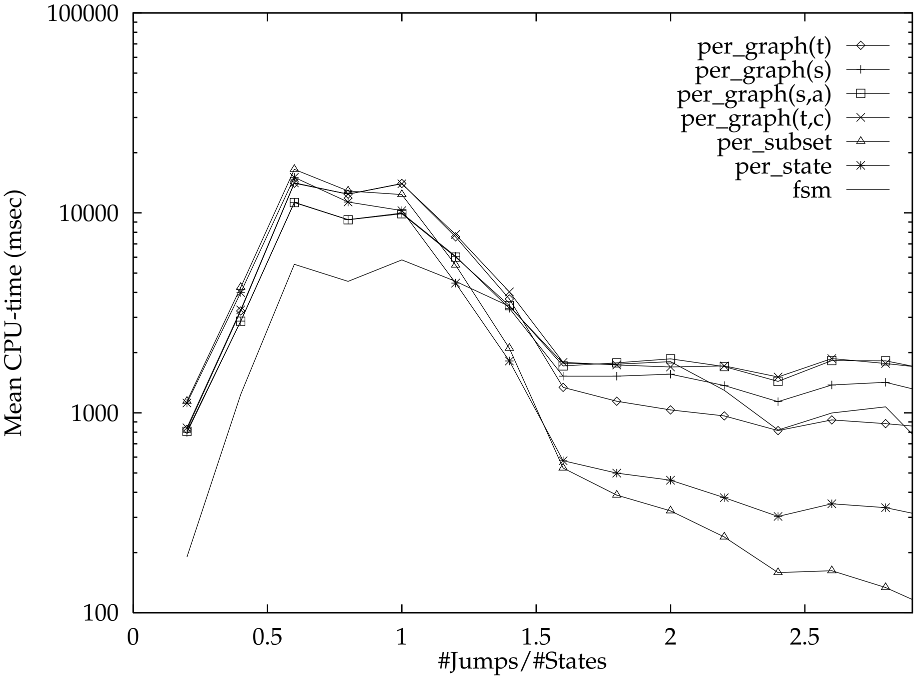

Figure 9
Average amount of CPU-time versus deterministic jump density for each of the algorithms, and FSM. Input automata have 100 states. Absolute transition densities: 0.001-0.0035.

Comparison with the FSM library We also provide the results for AT&T's FSM library again. FSM is designed to treat weighted automata for very general weight sets. The initial implementation of the library consisted of an on-the-fly computation of the epsilon-closures combined with determinisation. This was abandoned for two reasons: it could not be generalised to the case of general weight sets, and it was not outputting the intermediate epsilon-removed machine (which might be of interest in itself). In the current version ϵ-moves must be removed before determinisation is possible. This mechanism thus is comparable to our *per graph* variant. Apparently, FSM employs an algorithm equivalent to our *per graph*^{s,a}. The resulting determinised machines are generally larger than the machines produced by our integrated variants and the variants which incorporate ϵ-moves on the target side of transitions. The timings below are obtained for the pipe

```
fsmrmepsilon | fsmdeterminize
```

This is somewhat unfair since this includes the time to write and read the intermediate machine. Even so, it is interesting to note that the FSM library is a constant factor faster than our *per graph*^{s,a}; for larger numbers of jumps the *per state* and *per subset* variants consistently beat the FSM library.

Experiment: Automata generated by approximation algorithms The automata used in the previous experiments were randomly generated. 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. In particular, automata have been used which were created by Nederhof, using the technique described in Nederhof (1997). In addition, a small number of automata have been used which were created using the technique of Pereira and Wright (1997) (as implemented by Nederhof). We have restricted our attention to automata with at least 1000 states in the input.

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 the tables 1 and 2. One of the most striking examples is the *ygrim* automaton consisting of 3382 states and 9124 jumps. For this example, the *per graph* implementations ran out of memory (after a long time), whereas the implementation of the *per subset* algorithm produced the determinised automaton (containing only 9 states) within a single CPU-second. The FSM implementation took much longer for this example (whereas for many of the other examples it is faster than our implementations). Note that this example has the highest number of jumps per number of states ratio. This confirms the observation that the *per subset* algorithm performs better on inputs with a high deterministic jump density.

5 Conclusion

We have discussed a number of variants of the subset-construction algorithm for determinising finite automata containing ϵ-moves. The experiments support the following conclusions:

- The integrated variants *per subset* and *per state* work much better for automata containing a large number of ϵ-moves. The *per subset* variant tends to improve upon the *per state* algorithm if the number of ϵ-moves increases even further.

- We have identified four different variants of the *per graph* algorithm. In our experiments, the *per graph*^{t} is the algorithm of choice for automata

	Input			Output		
Id	#states	#trans	#jumps	#states		
				per graph^{s} per graph^{s,a} *FSM*	per graph^{t} per subset per state	per graph^{t,c}
g14	1048	403	1272	137	137	131
ovis4.n	1424	2210	517	164	133	107
g13	1441	1006	1272	337	337	329
rene2	1800	2597	96	846	844	844
ovis9.p	1868	2791	2688	2478	2478	1386
ygrim	3382	5422	9124	9	9	9
ygrim.p	48062	63704	109296	702	702	702
java19	54369	28333	51018	1971	1971	1855
java16	64210	43935	41305	3186	3186	3078
zovis3	88156	78895	68093	5174	5154	4182
zovis2	89832	80400	69377	6561	6541	5309

Table 1

The automata generated by approximation algorithms. The table lists the number of states, transitions and jumps of the input automaton, and the number of states of the determinised machine using respectively the *efree*^{s}, *efree*^{t}, and the *efree*^{t,c} variants.

	CPU-time (sec)						
	graph^{t}	graph^{t,c}	graph^{s}	graph^{s,a}	subset	state	*FSM*
g14	0.4	0.4	0.3	0.3	0.4	0.2	0.1
ovis4.n	0.9	1.1	0.8	1.0	0.7	0.6	0.6
g13	0.9	0.8	0.6	0.6	1.2	0.7	0.2
rene2	0.2	0.3	0.2	0.2	0.2	0.2	0.1
ovis9.p	36.6	16.0	16.9	17.0	25.2	20.8	21.9
ygrim	-	-	-	-	0.9	21.0	512.1
ygrim.p	-	-	-	-	562.1	-	4512.4
java19	55.5	67.4	52.6	45.0	25.8	19.0	3.8
java16	30.0	45.8	35.0	29.9	11.3	12.1	3.0
zovis3	741.1	557.5	-	407.4	358.4	302.5	325.6
zovis2	909.2	627.2	-	496.0	454.4	369.4	392.1

Table 2

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.

containing few ϵ moves, because it is faster than the other algorithms, and because it produces smaller automata than the *per graph*^{s} and *per graph*^{s,a} variants.

- The *per graph*^{t,c} variant is an interesting alternative in that it produces the smallest results. This variant should be used if the input automaton is expected to contain many non-co-accessible states.
- Automata produced by finite-state approximation techniques tend to contain many ϵ-moves. We found that for these automata the differences in speed between the various algorithms can be enormous. The *per subset* and *per state* algorithms are good candidates for this application.

We have attempted to characterize the expected efficiency of the various algorithms in terms of the number of jumps and the number of states in the input automaton. It is quite conceivable that other simple properties of the input automaton can be used even more effectively for this purpose. One reviewer suggests to use the number of strongly ϵ-connected components (the strongly connected components of the graph of all ϵ-moves) for this purpose. We leave this and other possibilities to a future occasion.

Acknowledgments
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. The comments of the anonymous FSMNLP and CL reviewers were extremely useful.

References

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

Black, A.W. 1989. Finite state machines from feature grammars. In *International Workshop on Parsing Technologies*, pages 277–285, Pittsburgh.

Chomsky, Noam. 1963. Formal properties of grammars. In R. Duncan Luce, Robert R. Bush, and Eugene Galanter, editors, *Handbook of Mathematical Psychology; Volume II*. John Wiley, pages 323–418.

Chomsky, Noam. 1964. On the notion 'rule of grammar'. In Jerry E. Fodor and Jerrold J. Katz, editors, *The Structure of Language; Readings in the Philosophy of Language*. Prentice Hall, pages 119–136.

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

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.

Gerdemann, Dale and Gertjan van Noord. 1999. Transducers from rewrite rules with backreferences. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*, Bergen Norway.

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

Johnson, 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.

Johnson, Mark. 1998. Finite-state approximation of constraint-based grammars using left-corner grammar transforms. In *COLING-ACL '98. 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics. Proceedings of the Conference*, Montreal.

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

Miller, George and Noam Chomsky. 1963. Finitary models of language users. In R. Luce, R. Bush, and E. Galanter, editors, *Handbook of Mathematical Psychology. Volume 2.* John Wiley.

Mohri, Mehryar, Fernando C.N. Pereira, and Michael Riley. 1998. A rational design for a weighted finite-state transducer library. In *Automata*

Implementation. Second International Workshop on Implementing Automata, WIA '97. Springer Verlag. Lecture Notes in Computer Science 1436.

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

Nederhof, Mark-Jan. 1998. Context-free parsing through regular approximation. In *Finite-state Methods in Natural Language Processing*, pages 13–24, Ankara.

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, pages 87–108. Lecture Notes in Computer Science 1260.

van Noord, Gertjan. 1998. The treatment of epsilon moves in subset construction. In *Finite-state Methods in Natural Language Processing*, Ankara. cmp-lg/9804003.

van Noord, Gertjan. 1999. FSA6 reference manual. The *FSA Utilities* toolbox is available free of charge under Gnu General Public License at http://www.let.rug.nl/˜vannoord/Fsa/.

van Noord, Gertjan and Dale Gerdemann. 1999. An extendible regular expression compiler for finite-state approaches in natural language processing. In O. Boldt, H. Juergensen, and L. Robbins, editors, *Workshop on Implementing Automata; WIA99 Pre-Proceedings*, Potsdam Germany.

O'Keefe, Richard A. 1990. *The Craft of Prolog*. The MIT Press, Cambridge Mass.

Pereira, Fernando C. N. and R. N. Wright. 1991. Finite-state approximation of phrase structure grammars. In *29th Annual Meeting of the Association for Computational Linguistics*, Berkeley.

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.

Rood, C.M. 1996. Efficient finite-state approximation of context free grammars. In A. Kornai, editor, *Extended Finite State Models of Language*, Proceedings of the ECAI'96 workshop, pages 58–64, Budapest University of Economic Sciences, Hungary.