FSA Utilities:
A Toolbox to Manipulate Finite-state Automata

Gertjan van Noord

Vakgroep Alfa-informatica & BCN
Rijksuniversiteit Groningen
vannoord@let.rug.nl

Abstract. This paper describes the *FSA Utilities* toolbox: a collection of utilities to manipulate finite-state automata and finite-state transducers. Manipulations include determinization (both for finite-state acceptors and finite-state transducers), minimization, composition, complementation, intersection, Kleene closure, etc. Furthermore, various visualization tools are available to browse finite-state automata. The toolbox is implemented in SICStus Prolog.

1 Introduction

This paper describes the *FSA Utilities* toolbox: a collection of utilities to manipulate finite-state automata and finite-state transducers. Manipulations include determinization (both for finite-state acceptors and finite-state transducers), minimization, composition, complementation, intersection, Kleene closure, etc. Furthermore, various visualization tools are available to browse finite-state automata. The toolbox is implemented in SICStus Prolog.

The motivation for the *FSA Utilities* toolbox has been the rapidly growing interest in finite-state techniques for *computational linguistics*. In particular, finite-state techniques are being used in computational phonology and morphology (Kaplan and Kay 1994), efficient dictionary lookup and part-of-speech tagging (Roche and Schabes 1995), natural-language parsing (Voutilainen and Tapanainen 1993), techniques for parsing ill-formed input (Lang 1989, van Noord 1995) and speech recognition (Oerder and Ney 1993, Pereira and Riley 1996), etc. The *FSA Utilities* toolbox has been developed to experiment with the techniques presented in these publications.

The following reasons for the popularity of finite-state techniques can be identified. Finite-state automata provide a well-studied, simple and very efficient formalism. Moreover, finite-state automata can be combined in various ways to construct larger automata from smaller ones. Such combined automata can still be processed very efficiently.

In this paper, I will illustrate the use of the *FSA Utilities* toolbox by means of a number of examples in section 3. I will then describe each of the operations which is provided by the *FSA Utilities* toolbox in more detail in section 4. Section 5 describes the visualization tools. Finally I will discuss some of the implementational issues in section 6. I first introduce the format for finite automata that is being used by *FSA Utilities*.

2 Representation

2.1 Finite-state Automata

A finite-state automaton is defined as a set of Prolog clauses. Some very limited familiarity with Prolog is assumed. A finite-state automaton is defined using the following relations:

`start(State).` `State` is the start state. There should be at least one start state. Multiple start states are supported.

`final(State).` `State` is a final state. There can be any number of final states.

`trans(State0,Sym,State).` There is a transition from `State0` to `State` with associated symbol `Sym`. There can be any number of transitions.

`jump(State0,State).` There is an ϵ-transition from `State0` to `State`. There can be any number of jumps.

It is assumed that states and symbols are ground Prolog terms (but see the discussion on constraints in section 2.3 below). A finite-state automaton defining the language consisting of any number of `a`'s over the alphabet {`a,b`} is defined as follows:

```
start(0).          final(0).          trans(0,a,0).
trans(0,b,1).      trans(1,a,1).      trans(1,b,1).
```
(1)

It is worthwhile to note that we do not require that the transition relation is total. In other words, there can be states that have no outgoing transitions for certain symbols. In such cases we assume that these transitions do exist, but lead to some non-final state from which you cannot leave. This state is sometimes called the *sink*. In example 1 state `1` is such a sink. Thus, we may abbreviate example 1 as

```
start(0).          final(0).          trans(0,a,0).
trans(0,b,1).
```
(2)

In the representation we are using, we do not require that the alphabet and the set of states is defined explicitly. If the previous example is further abbreviated as:

```
start(0).          final(0).          trans(0,a,0).
```
(3)

the information that `b` is part of the alphabet is lost. This is problematic if the complement of this language is to be constructed; for this reason the difference operation has been introduced (cf. section 4.2).

2.2 Finite-state Transducers

Finite-state transducers are represented using the same conventions we use for finite-state acceptors, except that the symbol part of a transition now is written as a pair `In/Out`, indicating respectively the symbol that is read and the symbol that is written.

Moreover, the symbol `'$E'` is used to indicate ϵ, in order to allow transitions in which the input or output symbol is the empty symbol. This implies that there are two ways to define a jump in a finite-state transducer: `trans(Q0,'$E'/'$E',Q)` is equivalent to `jump(Q0,Q)`. Note that the system does not allow the use of sequences of symbols in transitions (it can be shown that this does not restrict the transductions that can be defined).

A sub-sequential transducer is like a finite-state transducer except that we associate a sequence of symbols with a final state. These symbols are written to the output tape if the system halts in that final state. Such sub-sequential transducers are deterministic with respect to the 'in'-part of symbols: for each state there is at most one transition leaving that state upon reading some symbol. Note that we use sub-sequential transducers because the determinization of a finite-state transducer leads to a sub-sequential transducer (Roche and Schabes 1995). Sub-sequential transducers are represented as finite-state transducers, except that we now use a different predicate to indicate the final state with a sequence of symbols:

`final_td(State,Symbols)` indicates that `State` is a final state with associated sequence of symbols `Symbols`. `Symbols` is a ground Prolog list of symbols.

Sub-sequential transducers can be used wherever finite-state transducers are allowed.

2.3 Simple Prolog Constraints

Because finite-state automata are defined by Prolog clauses, it is very simple to attach Prolog constraints. For example, suppose that for a certain application it is useful to consider two subclasses of the alphabet, namely V (*vowels*) and C (*consonants*), we could then use the following technique to define the sequences $C^*V^*C^*$:

```
vowel(a).   vowel(e).   vowel(i).   vowel(o).   vowel(u).

cons(b).    cons(c).    cons(d).    cons(f).    cons(g).
cons(h).    cons(j).    cons(k).    cons(l).    cons(m).
cons(n).    cons(p).    cons(q).    cons(r).    cons(s).
cons(t).    cons(v).    cons(w).    cons(y).    cons(z).        (4)

start(0).   final(2).   jump(0,1).  jump(1,2).
trans(0,C,0) :- cons(C).
trans(1,V,1) :- vowel(V).
trans(2,C,2) :- cons(C).
```

I only allow constraints for which Prolog's built-in search procedure terminates. These Prolog constraints therefore do not increase the formal power of the formalism, although they are useful to define automata in a convenient and concise way.

3 Some examples

The *FSA Utilities* are available through a single UNIX command `fsa` which can take a number of different options. Suppose we have defined the following finite-state automaton, defining the language consisting of all the strings made up of an even number of `a`'s followed by an even number of `b`'s, in a file called `aabb.nd`:

```
start(0).          trans(0,a,1).      trans(1,a,0).
jump(0,2).         trans(2,b,3).      trans(3,b,2).
final(2).
```
(5)

In order to determinize this automaton, we give the following UNIX command (note that in these examples lines starting with a `%` are typed to the UNIX shell; the lines that follow are output of the *FSA Utilities* program):

```
% fsa -d <aabb.nd >aabb.d
```
(6)

This command writes the determinized automaton to the file `aabb.d`. This file now contains:

```
start(q0).         final(q0).         final(q1).
trans(q0,b,q2).    trans(q0,a,q3).    trans(q2,b,q1).
trans(q1,b,q2).    trans(q3,a,q0).
```
(7)

It is also possible to create an automaton on the basis of a regular expression. The file aabb.d could also be obtained by the command:

```
fsa -r '(a.a)* .(b.b)*' | fsa -d > aabb.d
```
(8)

In such regular expressions the dot (`.`) is used to indicate concatenation, union is defined by the semi-colon (`;`) and Kleene closure is defined by the asterisk (`*`).

A minimized automaton is obtained with the `-m` option. We can, for example, pipe the result of determinization to another incarnation of the fsa command. The following pipe produces the minimal finite-state acceptor for the language consisting of an even number of `a`'s followed by an uneven number of `b`'s:

```
% fsa -r '(a.a)* .(b.b)* .b' | fsa -d | fsa -m
start(q0).         final(q1).
trans(q0,a,q2).    trans(q0,b,q1).    trans(q1,b,q3).
trans(q2,a,q0).    trans(q3,b,q1).
```
(9)

Next consider a finite-state *transducer* which copies its input (strings of `a`'s and `b`'s) to its output, except that if an `a` is followed by a `b`, then this `a` becomes a `b`. Suppose this transducer is defined in the file a2b.tnd as follows:

```
start(0).             trans(0,b/b,0).       trans(0,a/b,1).
trans(1,b/b,0).       trans(0,a/a,2).       trans(2,a/b,1).       (10)
trans(2,a/a,2).       final(0).             final(2).
```

Such a transducer can be determinized (using the algorithm described in Roche and Schabes (1995) with the command:

```
% fsa -td <a2b.tnd >a2b.td                                        (11)
```

The file a2b.td now contains:

```
start(q0).            final_td(q0,[]).      final_td(q1,[a]).
trans(q0,b/b,q0).     trans(q0,a/'$E',q1). trans(q1,b/b,q2).     (12)
trans(q2,'$E'/b,q0). trans(q1,a/a,q1).
```

In order to use this transducer to transduce strings, it may be worthwhile to compile it into an efficient Prolog program implementing the transduction. The Prolog program will be fully deterministic; using first argument indexing, this determinism is visible to modern Prolog compilers. This implies that transduction is computed in linear time (with respect to the size of the input), and is independent of the size of the transducer.

```
% fsa -ct <a2b.td >a2b.pl                                         (13)
```

The file `a2b.pl` contains:

```
t_accepts(S,T) :- t_accepts_q0(S,T).
t_accepts_q0([],[]).
t_accepts_q1([],[a]).
t_accepts_q0([S|T],O) :- t_accepts_q0(S,T,O).
t_accepts_q1([S|T],O) :- t_accepts_q1(S,T,O).
t_accepts_q2([S|T],O) :- t_accepts_q2(S,T,O).                     (14)
t_accepts_q0(b,T,[b|O]) :- t_accepts_q0(T,O).
t_accepts_q0(a,T,O) :- t_accepts_q1(T,O).
t_accepts_q1(b,T,[b|O]) :- t_accepts_q2(T,O).
t_accepts_q1(a,T,[a|O]) :- t_accepts_q1(T,O).
t_accepts_q2(S,[b|O]) :- t_accepts_q0(S,O).
```

This Prolog program can be used to transduce the string `aabbc` by issuing the command:

```
% fsa -transduce a a b b < a2b.pl                                 (15)
a b b b
```

In regular expressions it is possible to use pairs of symbols in order to obtain finite-state transducers. Thus we can have examples such as the following:

```
% fsa -r '((a:a;b:b;c:c)* .(d:e)* .(a:a;b:b;c:c)*)*' \
       | fsa -td                                                  (16)
start(q0).            final_td(q0,[]).      trans(q0,d/e,q0).
trans(q0,c/c,q0).     trans(q0,b/b,q0).     trans(q0,a/a,q0).
```

As another example, it is possible to compose `aabb.nd` with the transducer `a2b.tnd`. The following pipe produces the minimized, determinized composition:

```
fsa -compose_fsa aabb.nd a2b.tnd | fsa -d \
                           | fsa -m > result.d
```
(17)

Such automata can be inspected with the Tk Widget (presented in section 5), for example, as in figure 1.

In this paper the use of the *FSA Utilities* toolbox is illustrated by means of UNIX commands. However, this is not the only possible way of using the toolbox. Care has been taken to implement the toolbox in such a way that it is straightforward to use each of the operations as a Prolog library. For this reason most of the operations are defined in separate modules.

4 Operations on Finite Automata

The FSA Utilities toolbox provides a number of operations on finite-state automata. These operations are presented in this section. First I present the operations for finite-state acceptors, then I define the operations related to finite-state transducers.

4.1 Finite-state Acceptors

The `-accepts Words` option can be used to check whether a given string of symbols (`Words`) is accepted by a given finite state automaton (read from standard input). This automaton is defined according to the conventions discussed in section 2. Alternatively this file is a compiled finite-state automaton as produced by the `-compile` option presented below. The program determines whether `Words` is accepted by this automaton or not. If it is, the program returns successfully; otherwise it exits with exit code 1. Note that this procedure is only guaranteed to be linear in the length of the input string if the finite state automaton is both deterministic and compiled. In the case of such a compiled deterministic automaton SICStus Prolog's first argument indexing is exploited to ensure that acceptance is also independent of the size of the automaton.

The `-produce` option is used to generate all possible strings accepted by the input automaton. Strings are produced in increasing length. Clearly this operation need not terminate if the automaton defines a language consisting of an unbounded number of strings.

Consider the following examples.

```
% fsa -a a a a a b b < aabb.nd
yes
% fsa -a a a a b b < aabb.nd
no
% fsa -d <aabb.nd | fsa -c > aabb.pl
% fsa -a a a a a b b b b b b < aabb.pl
yes
```
(18)

```
% fsa -p <aabb.nd

a a
b b
a a a a
a a b b
b b b b
a a a a a a
a a a a b b
a a b b b b
b b b b b b
...
```
(19)

4.2 Regular expressions

The following options can be used to construct automata for regular languages on the basis of the language(s) defined by one or more input automata:

```
-concat File1 File2
-kleene File1
-union File1 File2
-reverse File1
-intersect File1 File2
-complement File1
-difference File1 File2
```

These options can be used to combine automata by concatenation, Kleene closure, union, reversal, intersection, complementation or difference respectively. Examples:

```
% fsa -complement <aabb.d
start(q4).          final(q5).          final(q6).
final(q7).          trans(q5,a,q5).     trans(q5,b,q5).
trans(q4,a,q7).     trans(q4,b,q6).     trans(q8,a,q5).
trans(q8,b,q6).     trans(q6,a,q5).     trans(q6,b,q8).
trans(q7,a,q4).     trans(q7,b,q5).
```
(20)

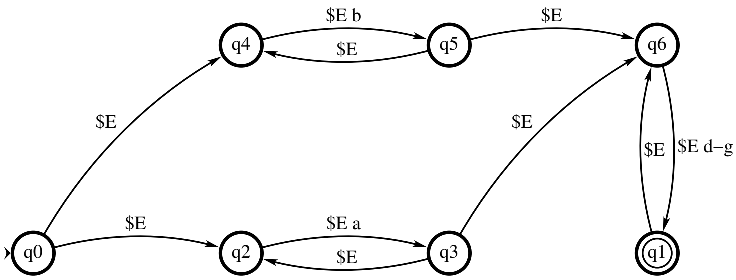
Because the system assumes that the alphabet consists only of the symbols occurring in the input automaton, it may be safer to use the `-difference` option. In this case an automaton is written to standard output, defining all strings that are defined by the automaton in File1 except for the strings defined in File2. For example, if the file `abc.nd` consists of the automaton defining all strings over the alphabet {a,b,c}, then we get:

```
% fsa -difference abc.nd aabb.nd
start(q9).            final(q10).           final(q11).
final(q12).           trans(q9,a,q12).      trans(q9,b,q11).
trans(q9,c,q10).      trans(q10,a,q10).     trans(q10,b,q10).
trans(q10,c,q10).     trans(q11,a,q10).     trans(q11,b,q13).
trans(q11,c,q10).     trans(q12,a,q9).      trans(q12,b,q10).
trans(q12,c,q10).     trans(q13,a,q10).     trans(q13,b,q11).
trans(q13,c,q10).
```
(21)

The system also supports the construction of an automaton on the basis of a regular expression: the option `-r Expression` can be used. The operators that are allowed are Kleene closure (`*`), concatenation (`.`) and union (`;`). Moreover, it is possible to have transduction pairs (`:`). Finally, the system understands intervals. The expression `a-z` is a shorthand for the expression `a;b;c;..;z`. For example:

```
% fsa -r '(a*;b*).(d-g)*' | fsa -ps
```
(22)

produces the following postscript picture:

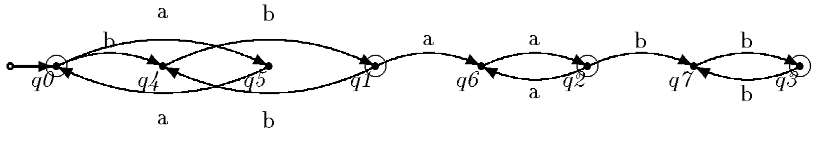


4.3 Determinization

The `-d` option is used to determinize a finite state automaton. Standard input consists of the finite-state automaton to be determinized. The determinized finite-state automaton is written to standard output. The determinization algorithm uses a version of the subset construction (Hopcroft and Ullman 1979) in which only those subsets are considered that can be reached from an initial state. Example:

```
% fsa -d aabbaabb.nd | fsa -tex -q 0 > aabbaabb.pic
```
(23)

This results in the following LaTeX picture of the determinized automaton:



4.4 Minimization

Currently, the toolbox supports three different implementations of the minimization operation. The option `-m` indicates that the algorithm of Hopcroft and Ullman (1979) should be used. The input automaton should already be deterministic. The option `-m2` indicates the use of Brzozowski's method (Brzozowski 1962). The input automaton does not need to be deterministic. The option `-m3` indicates that Hopcroft's algorithm should be used (Hopcroft 1971). The input automaton must be deterministic. The implementation of these algorithms is described in section 6.3.

```
% fsa -d < y.nd | fsa -m > y.m
% fsa -m2 < y.nd > y.m
```
(24)

4.5 Finite-state Transducers

If the option `-transduce Words` is used, the system produces all possible transductions of `Words` on the basis of the finite-state transducer read from standard input. This transducer can be specified in compiled format (a .pl file resulting from the `-ct` option presented below).

```
% fsa -transduce a a b b <a2b.td
a b b b
```
(25)

If multiple output strings are possible, then all strings are produced. Consider the `-compose_string` option (discussed below) as an alternative in such cases: this produces a single automaton.

If the option `-ta File` is used, the system reads lines from standard input and writes for each line the line(s) as produced by the transducer (in File) to standard output.

If the option `-tp` is used, a finite state transducer is read from standard input. This transducer need not be deterministic. All pairs of sentences in the mapping defined by the transducer are written to standard output. Pairs are produced in increasing length of the input string.

```
% fsa -td <a2b.a2b.tnd |fsa -tp
==>                          b ==> b
a ==> a                      b b ==> b b
b a ==> b a                  a b ==> b b
a a ==> a a                  b b b ==> b b b
b b a ==> b b a              b a b ==> b b b
b a a ==> b a a              a b b ==> b b b
a b a ==> b b a              a a b ==> b b b
...
```
(26)

The `-ct` option is used to compile a transducer into a set of Prolog clauses (similar to the `-compile` option for acceptors). Standard input consists of the finite-state automaton to be compiled. The compiled clauses are written to standard output. If the input is a deterministic transducer, then the compiled clauses can be used by Prolog deterministically for transduction (clauses are indexed such that the Prolog compiler is able to recognize that the application of each clause is deterministic); transduction time is then independent of the size of the automaton. Examples:

```
% fsa -ct <a2b.td >a2b.pl
```

(27)

The options `-domain` and `-range` can be used to obtain the domain (range) of the mapping defined by the transducer read from standard input. The `-identity` option can be used to obtain a transducer defining the identity relation for the language defined by the finite-state automaton read from standard input:

```
% fsa -domain <a2b.a2b.tnd |fsa -d |fsa -m
start(q0).           final(q0).           trans(q0,a,q0).
trans(q0,b,q0).
% fsa -identity <aabb.nd
start(q0).           final(q1).           trans(q0,a/a,q2).
trans(q2,a/a,q0).    trans(q1,b/b,q3).    trans(q3,b/b,q1).
jump(q0,q1).
```

(28)

4.6 Composition

Transducers can be composed with the option `-compose File1 File2`. In this form, the two files are assumed to contain finite state transducers. The (serial) composition is written to standard output. In the form `-compose_fsa File1 File2`, the first file contains a finite state automaton and the second file contains a finite state transducer. The composition is written to standard output. In the form `-compose_string Words`, a finite-state transducer is read from standard input and composed with the (finite-state automaton defining the language consisting of the single) string Words. The composition is written to standard output. Examples:

```
% fsa -compose a2b.tnd a2b.tnd >a2b.a2b.tnd
% fsa -compose_fsa aabb.nd a2b.tnd
% fsa -compose_string a a b <a2b.tnd |fsa -d |fsa -m
start(q0).           final(q1).           trans(q0,a,q2).
trans(q2,b,q3).      trans(q3,b,q1).
```

(29)

Note that the `compose_string` operation differs from the `transduce` operation discussed previously. If the transducer is non-deterministic and relates `Words` to several output strings, then this option constructs a finite-state automaton defining exactly the set of those output strings; if the `transduce` option is used, then all of the resulting strings are written to standard output.

In computational morphology a different kind of composition is sometimes identified: parallel composition. The parallel composition of two transducers is obtained by the `-intersect` option introduced above for finite state acceptors.

4.7 Determinization of Transducers

Determinization of a transducer can be performed with the `-td` option. Standard input consists of the transducer to be determinized. The determinized transducer is written to standard output.

```
% fsa -td <a2b.a2b.tnd >a2b.a2b.td                    (30)
```

The determinization algorithm for transducers is described in Roche and Schabes (1995). Not every transducer can be determinized. This procedure is guaranteed to terminate only if the input transducer *can* be determinized. Furthermore, note that the output transducer is a *sub-sequential* transducer: each final state is associated with a sequence of symbols that are outputted if the system halts in that state.

5 Visualization of Automata

The system offers a number of different means of visualization. Representations of finite-state automata can be constructed which are compatible with either the VCG (Sander 1995) or the daVinci (Fröhlich and Werner 1994) visualization programs. These programs are generic graph-visualization programs; the results for finite-state automata tend to differ a lot from example to example. VCG has no problems with larger graphs; daVinci sometimes has. Various options can be used to obtain different effects.

Furthermore, the system provides its own visualization through a Tk-widget (Ousterhout 1994). This widget allows dragging of states to manipulate the visualization of an automaton interactively. Moreover, LaTeX picture output or Postscript output can be generated on the basis of this (interactively manipulated) visualization. The system can also output LaTeX picture output or Postscript output directly, using the `-tex` and `-ps` options.

5.1 VCG

If the option `-vcg` is used, a representation compatible with the VCG 1.30 graph-visualization program is written to standard output, on the basis of the finite-state automaton read in from standard input. For example, if the file fig19.tnd consists of the transducer given in figure 19 of Roche and Schabes (1995), then the following pipe produces the illustration given in figure 2.

```
% fsa -td fig19.tnd | fsa -vcg | xvcg -                    (31)
```

5.2 DaVinci

If the `-davinci` option is used, a term-representation compatible with the requirements of the daVinci 2.0 graph-visualization program is written to standard output, on the basis of the finite-state automaton read in from standard input. An example of daVinci is given in figure 3.

5.3 Latex Picture

The `-tex` option is used to produce a LaTeX picture. The same visualization mechanism is used as for the `-tk` option, cf. below. Also, the same options to influence this visualization can be given.

`% fsa -tex <ba_n.nd >ba_n.pic` (32)

5.4 Postscript

The option `-ps` is similar to the `-tex` option. In this case the output is given as a Postscript picture.

5.5 The Tk Widget

The option `-tk [[-q Integer] [-xd Dist] [-angle Angle]] File` indicates that a finite state automaton is read from File and shown in a Tk Widget. The `-xd` option can be used to alter the default X distance of nodes (default: 120). The `-angle` option indicates the angle of edges (default: 0.25). Straight lines can be obtained with a value of 0. The quality option indicates how hard the system should try to obtain a good visualization. Default is 1, which indicates that not much effort is made at reducing the number of crossing branches; larger integers will (dramatically) increase processing time, resulting sometimes in slightly better output.

The Tk Widget offers a graphical user interface to browse finite-state automata. For an illustration, consider figure 1. The use of this widget is explained now as follows.

The Canvas The canvas displays the finite-state automaton. The display can be controlled by a number of options to the `fsa` command, or on-line by using the button-bar.

By default, edges are not drawn as straight lines, but as curves. The angle of the edge is determined by the `-angle` option or the `Edge Angle` field on the button-bar.

The default distance between nodes is determined by the `-xd` option or the `X-distance` field on the button-bar.

Finally the system provides limited functionality to compute the most suitable geometry. The intent is to minimize the number of crossing edges, as well as the size of the area needed to display an automaton. Since in general such a procedure is much too costly, the `-q` option or the `Quality` field on the button-bar can be used to tell the system how much effort should be spent on this. A higher value should imply better quality.

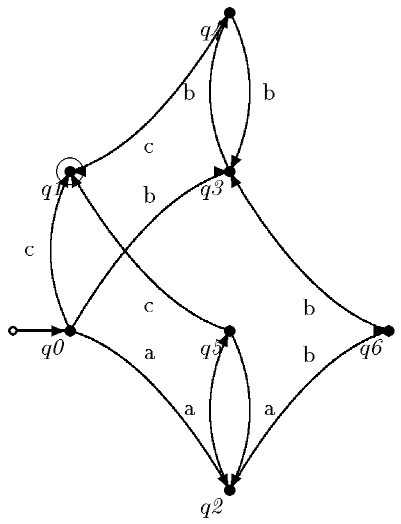
The finite-state automaton can be manipulated manually by dragging states of the automaton to a new place. This possibility implies that you can adapt the visualization of the automaton to your own taste, and to the characteristics of a particular automaton.

The Button-bar

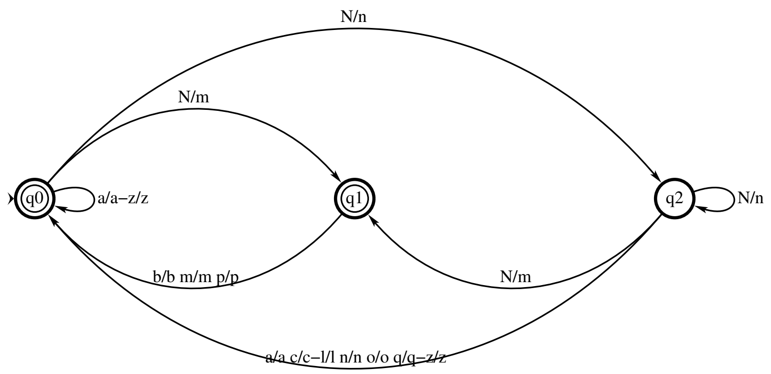
Save The current geometry of the finite-state automaton is saved in the file from which the current automaton was read. Note that the next time this file is used as input for the `fsa -tk` command, all geometry options are ignored, and this information is used instead.

SaveAs As the above, but the user is prompted for a file-name.

SaveAsTeX The user is prompted for a file-name. In this file the system outputs the current geometry of the automaton as a LaTeX picture. This can then be used in a LaTeX file, to give e.g.:



SaveAsPs The user is prompted for a file-name. In this file the system outputs the current geometry of the automaton as a Postscript picture.



Load You are prompted for a file-name. The file-name is assumed to contain a finite-state automaton which will be displayed in the canvas.

Redraw The current automaton is redrawn.

Revert All changes are discarded and the automaton is redrawn.

Pause The Tk-Widget is removed from the screen.

Debugging When this option is set, progress information during the computation of the geometry is shown.

Quit The application is halted.

Edge Angle Determines the edge of an angle. When this is set to 0 then straight lines are used. Useful values are between 0 and 1.

X-distance Determines the distance between nodes.

Quality Determines the effort that should be spent on the computation of the geometry. If this value is larger than 0, the system tries to minimize the number of crossing edges and the size of the area needed for the automaton. Larger values dramatically increase the time needed to compute this.

Help The man page is displayed in an `xterm`.

6 A note on Implementation

Prolog is a high-level language in which operations on finite-state automata can be implemented rather straightforwardly.

However, Prolog lacks the appropriate data-structures to implement e.g. the subset-construction algorithm as efficiently as possible. For this reason I have developed a library which implements a data-structure which approximates a hash-table: this data-structure provides *log* time access rather than constant time access.

Furthermore, note that Prolog's built-in search method (backtracking) is not used in the implementation of the determinization and minimization operations. These operations are implemented by deterministic algorithms. If we are careful in designing the Prolog code for these algorithms, then the resulting Prolog code can be run efficiently, because modern Prolog compilers are capable of recognising determinism.

6.1 Determinization

The determinization algorithm I used is a subset-construction algorithm. For a given automaton (the 'input' automaton) the construction computes an equivalent, but deterministic automaton (the 'output' automaton). The algorithm maintains an agenda of states (of the output automaton) whose successors need to be computed, a table of states (of the output automaton) that have already been processed, a list of transitions (of the output automaton), and a list of final states (of the output automaton). A state of the output automaton is a subset of states from the input automaton. For each state on the agenda, the procedure computes all transitions that are possible leaving that state. If new states arise, then these states are added to the agenda.

The relation treat_start_states computes the set of states reachable without input consumption from the start states of the input automaton. This set of states from the input automaton constitutes the single start state of the output automaton. This start state is added to the list of final states if one of the input states is a final state. The initial agenda for the subset construction process contains this start state as a single element.

```
determinize([Start],Finals,Transitions) :-
    treat_start_states(Start,Table,Finals0,Finals),           (33)
    closure([Start],Table,Finals0,Transitions).
```

The relation closure recursively processes the agenda. Its arguments are respectively the list of states to be processed, the table of states already processed, the resulting list of final states, and the resulting list of transitions. If the agenda is empty, then the process finishes. Otherwise the first state of the agenda is considered and all transitions leaving this state are computed. Any new states that result are added to the agenda.

```
closure([],_table,[],[]). % no more states, finished.
closure([H|Agenda0],Table0,Tr,Fin) :-
    findall(NTrans,new_transition(H,NTrans),Transs),          (34)
    add(Transs,Agenda0,Agenda,Table0,Table,Tr0,Tr,Fin0,Fin),
    closure(Agenda,Table,Tr0,Fin0).
```

The relation new_transition defines what a possible transition is. The relation consists of a state (from the output automaton) and all transitions that are possible with this state as a source state.

```
new_transition(Begins,trans(Begins,Sym,Ends)) :-
    setof(End,a_trans(Begins,Sym,End),Ends).                  (35)
```

```
a_trans(Begins,Sym,End) :-
    member(Begin,Begins),
    trans(Begin,Sym,Mid),                                     (36)
    p_tr_jump(Mid,End).   % End is jump-reachable from Mid
```

The p_tr_jump relation consists of pairs of states from the input automaton such that the second state can be reached without reading any symbols from the first state (reflexive and transitive closure of the jump relation).

The relation add/9 inspects the given list of transitions for new target states. These transitions are added to the list of transitions. Each new state (i.e. not yet in the table) is added to the agenda and to the table (and to the list of final states if this state contains a final state of the input automaton). This concludes the presentation of the determinization algorithm.

6.2 Determinization of Transducers

The determinization algorithm that is used for transducers is a generalization of the algorithm presented. It is based on the algorithm described in Roche and Schabes (1995). In this generalized version, a state in the determinized transducer is a set of states from the input transducer, where each of these states is associated with a sequence of symbols that still need to be written.

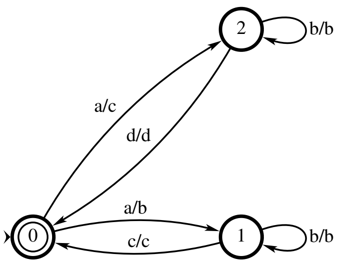
The output of this algorithm is a sub-sequential transducer. We can think of such a transducer as an ordinary transducer, except that a sequence of symbols is associated with each final state. If the system halts in a final state, then the associated symbols are written to the output tape. Such transducers are able to recognize whether or not the end of the string has been reached. Consider, for example, the following transducer. This transducer transduces a string of `a`'s into `b`'s, except for the last `a`:

```
start(q0).          final(q1).
trans(q0,a/b,q2).   trans(q2,a/a,q1).          (37)
jump(q0,q2).        jump(q2,q0).
```

In order to determinize this transducer, (i.e. ensure that for each state and each symbol there is only a single transition), the extra power of sub-sequential transducers is necessary:

```
start(q0).            final_td(q1,[a]).
trans(q0,a/'$E',q1).  trans(q1,a/b,q1).        (38)
```

Even if the input transducer defines a function, it is not always possible to construct such a deterministic (subsequential) transducer. An example of an inherently non-deterministic transducer is given below:



If the system is in state `0` and it reads an `a`, then it can only decide which transition to take after reading an unbounded number of `b`'s. The algorithm runs into an infinite loop for transducers for which no determinized version exists. [1]

6.3 Minimization

The *FSA Utilities* toolbox provides implementations of three different minimization algorithms. For a comparison of these and other minimization algorithms, refer to Watson (1995).

[1] Note that the results discussed in paragraph IV.6 of Berstel (1979), based on (Choffrut 1977), suggest that it is decidable whether an input transducer can be determinized.

The first minimization algorithm is taken from Hopcroft and Ullman (1979). In our implementation the following phases can be identified. Firstly, transitions are added to the incoming automaton to ensure that the transition function is total (note that it is assumed that the input is already deterministic). Then, in the second step, all pairs of non-identical states are computed. In the third phase, the automaton is adapted in such a way that states that were found to be identical receive a single name. Finally, some unneccessary transitions are thrown away (those ending and starting in the 'sink' state).

The computation of the set of pairs of non-identical states starts out with pairs of states for which one element is a final state and the other element is not a final state: these clearly are non-identical. These pairs are taken as an agenda and we then proceed to find more pairs of non-identical states on the basis of these pairs. Such new non-identical pairs can be obtained by the following reasoning. If `P` and `Q` are non-identical states, and there are transitions `trans(P1,Sym,P)` and `trans(Q1,Sym,Q)` then `P1, Q1` is a pair of non-identical states too.

The *FSA Utilities* toolbox also provides an implementation of Brzozowski's method of minimizing an automaton (Brzozowski 1962, Watson 1995). In this method the (possibly non-deterministic) automaton is reversed, determinized, reversed and determinized. Note that this method is often much faster than the previous method. Moreover the input can be non-deterministic. Finally, the implementation is very simple, since it consists only of automaton reversal (a particularly simple operation) and determinization (which is useful independently).

The third algorithm for minimization that is provided is the algorithm from Hopcroft (1971). Our implementation closely follows the presentation of Watson 1995). A disadvantage of this algorithm is that it is much more complicated than the other two.

6.4 Practical Experience

The main purpose of the development of the *FSA Utilities* toolbox has been to experiment with new techniques, rather than the construction of practical programs. However, one might still be interested in the practical efficiency of some of the operations provided. The determinization and minimization operations are the most critical operations. Minimization typically is much slower than determinization. The second version of the minimizer (using reversal and determinization twice) is much faster than the other versions. The current implementation of the determinizer runs much faster than a previous version, in which the transitive closure of jumps was computed as a separate pre-processing step. We also experienced that the second version of the minimizer runs much faster on a determinized automaton.

To get at least some idea of the system's practical behavior, first consider a finite-state automaton recognizing all mono-syllabic Dutch words. This non-deterministic automaton (written by Gosse Bouma) has 185 transitions, 9 jumps, 42 states and 26 symbols. Determinization takes 200 milliseconds (on a 1995 HP-UX 9000/735), resulting in an automaton with 49 states and 392 transitions. Minimization is much slower: it takes about 24 seconds to reduce the number of states to 36 and the number of transitions to 315. If Brzozowski's method is used, minimization can be done in less than three seconds and has the additional advantage that the input need not be determinized first. Hopcroft's method takes about 11 seconds.

As another example consider a finite-state *transducer* which translates temporal expressions such as *two minutes before half past seven* into `7 28`. This automaton is larger: it has 5824 states, 8809 transitions, 67 input symbols and 61 output symbols. Determinization takes less than 5 seconds. The result has 1714 states and 3362 transitions. In comparison, in order to determinize the automaton defining the domain of the transduction (obtained by simply removing the output part of the symbols) takes less than three seconds. The resulting automaton can be minimized (using Brzozowski's method) in about four seconds.

Availability

The *FSA Utilities* toolbox is available under GNU General Public License from anonymous ftp. Alternatively you can obtain the toolbox via the World-wide Web. Addresses:

```
ftp://ftp.let.rug.nl/pub/prolog-app/FSA/
http://www.let.rug.nl/~vannoord/Fsa/
```

Acknowledgements

Feedback from Gosse Bouma, Mark-Jan Nederhof, John Nerbonne, Fernando Pereira, Edwin Kuipers and anonymous reviewers from the First International Workshop on Implementing Automata is gratefully acknowledged. The option to produce postscript output is based on work by Peter Kleiweg.

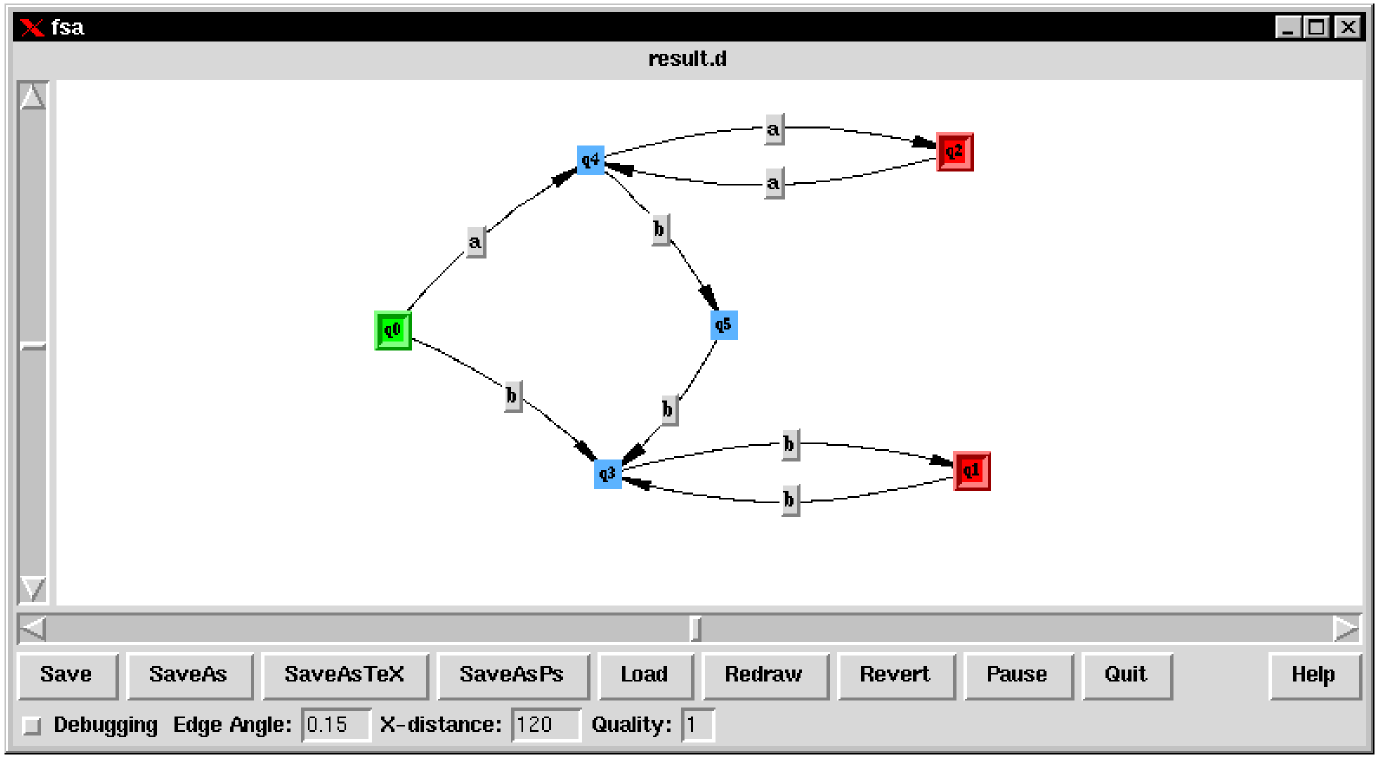


Fig. 1. The *FSA Utilities* toolbox uses Tcl/Tk for its built-in visualization of finite-state automata. The user can interactively alter the visualization by dragging states of the automaton to different positions. LaTeX picture output and Postscript output can be generated on the basis of the current view.

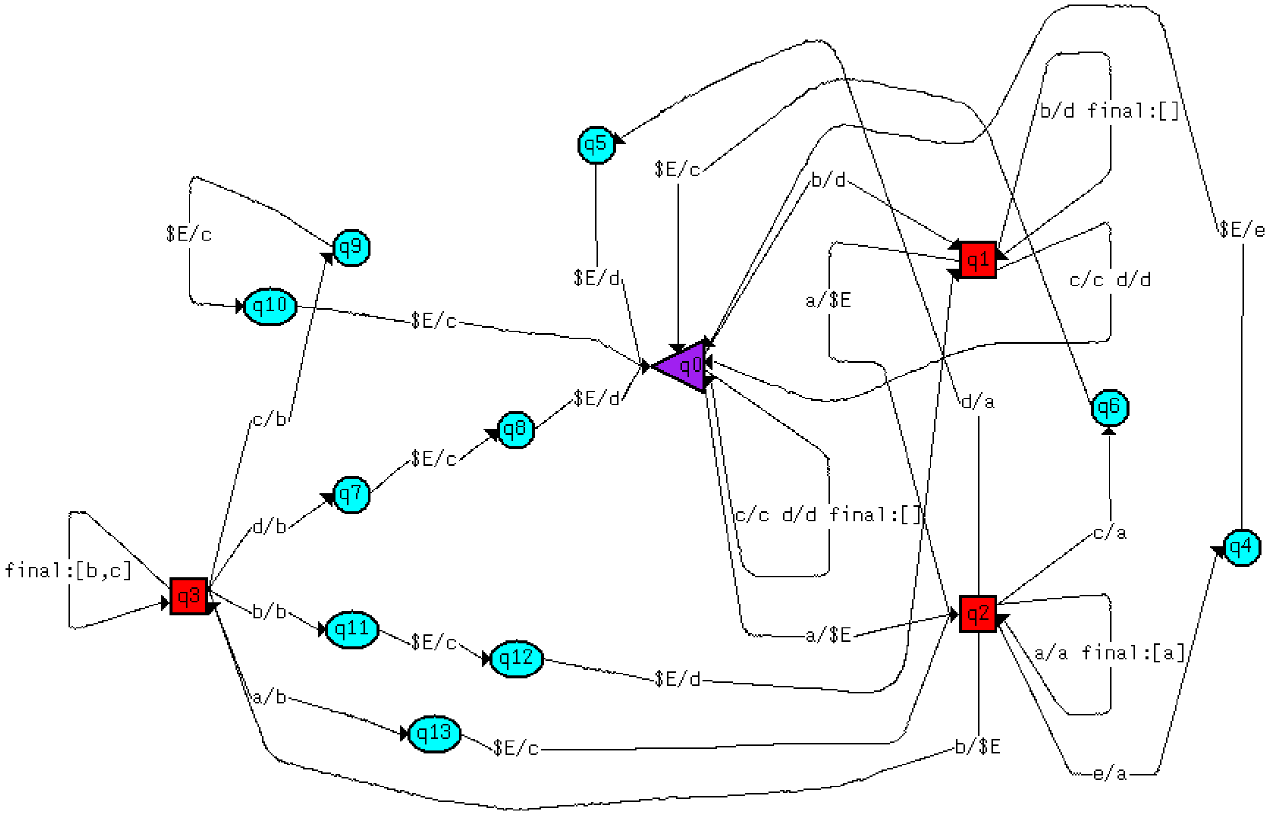


Fig. 2. The *FSA Utilities* program supports output for two external graph visualization programs. This figure illustrates the use of VCG. This example is the determinized version of an example given by Roche and Schabes (their figure 19).

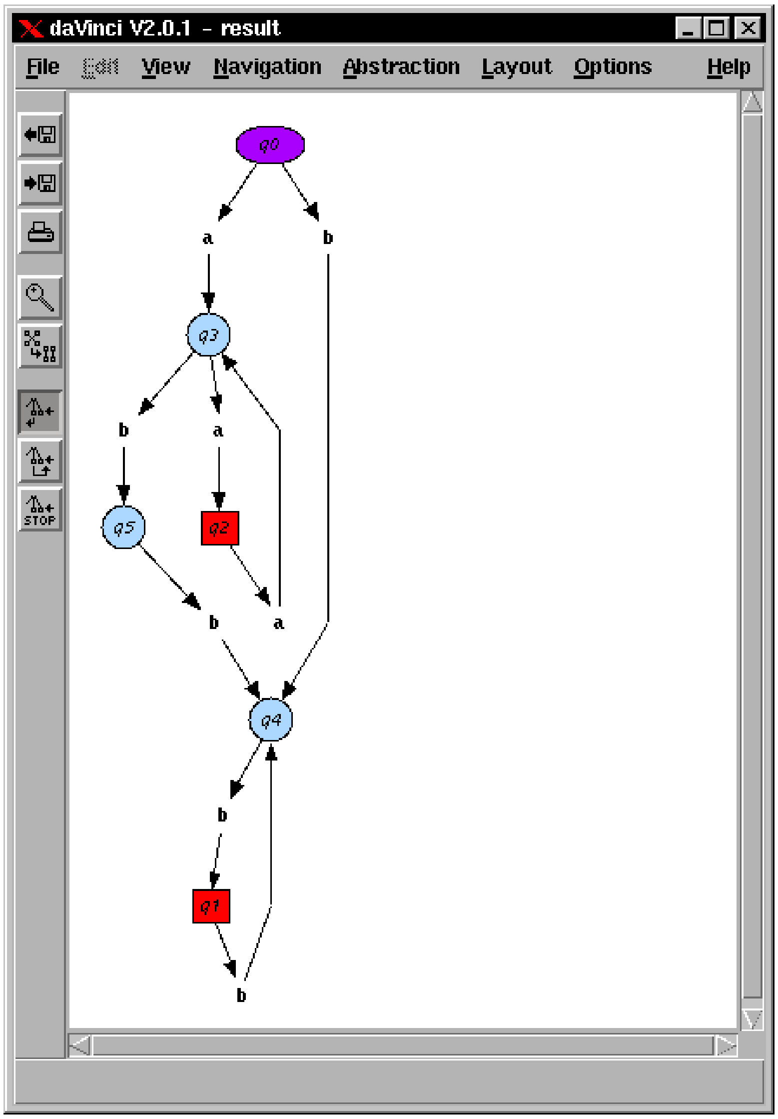


Fig. 3. The *FSA Utilities* toolbox is able to export finite-state automata in a representation suitable for external visualization tools. This figure illustrates the use of the daVinci visualization program to inspect a finite state automaton. Different colors and shapes are used to indicate the start states and the final states.

References

Berstel, Jean 1979. *Transductions and Context-Free Languages.* Teubner Studienbücher, Stuttgart.

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.

Choffrut, Ch. 1977 Une caractérisation des fonctions séquentielles et des fonctions sous-séquentielles en tant que relations rationelles. *Theoretical Computer Science* **5**, 325-338.

Fröhlich, M. and M. Werner. 1994. The graph visualization system *davinci* - a user interface for applications. Technical Report 5/94, Department of Computer Science; University of Bremen. Available by anonymous ftp: ftp://ftp.Uni-Bremen.DE/pub/graphics/daVinci/papers/techrep0594.ps.gz; cf. also http://www.informatik.uni-bremen.de/~inform/forschung/daVinci/.

Hopcroft, John E. 1971. An n log n algorithm for minimizing the states in a finite automaton. In: Z. Kohavi, editors, *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.

Kaplan, Ronald M. and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.

Lang, Bernard. 1989. A generative view of ill-formed input processing. In *ATR Symposium on Basic Research for Telephone Interpretation (ASTI)*, Kyoto Japan.

van Noord, Gertjan. 1995. The intersection of finite state automata and definite clause grammars. In *33th Annual Meeting of the Association for Computational Linguistics*, MIT Boston. Available from http://www.let.rug.nl/~vannoord/papers/.

Oerder, Martin and Hermann Ney. 1993. Word graphs: An efficient interface between continuous-speech recognition and language understanding. In *ICASSP Volume 2*, pages 119–122.

Ousterhout, John K. 1994. *Tcl and the Tk Toolkit.* Addison Wesley.

Pereira, Fernando C. N. and Michael D. Riley. 1996. Speech recognition by composition of weighted finite automata. Available as cmp-lg/9603001 from http://xxx.lanl.gov/cmp-lg.

Roche, Emmanuel and Yves Schabes. 1995. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21(2).

Sander, G. 1995. Graph layout through the VCG tool. In R. Tamassia and I.G. Tollis, editors, *Graph Drawing, DIMACS International Workshop GD '94, Proceedings; Lecture Notes in Computer Science 894.* Springer Verlag, pages 194–205. cf. also http://www.cs.uni-sb.de/RW/users/sander/html/gsvcg1.html.

Voutilainen, Atro and Pasi Tapanainen. 1993. Ambiguity resolution in a reductionist parser. In *Sixth Conference of the European Chapter of the Association for Computational Linguistics*, Utrecht.

Watson, Bruce W. 1995. *Taxonomies and Toolkits of Regular Language Algorithms.* Proefschrift Eindhoven University of Technology.