Automata

A package on automata

Version 1.14

Manuel Delgado

Steve Linton

José João Morais

Manuel Delgado
Email: mdelgado@fc.up.pt
Homepage: https://cmup.fc.up.pt/cmup/mdelgado/
Address: Departamento de Matemática - Faculdade de Ciências
Porto
Portugal

Steve Linton
Email: steve.linton@st-andrews.ac.uk
Homepage: http://www-groups.dcs.st-and.ac.uk/~sal/
Address: School of Computer Science
University of St Andrews
Jack Cole Building, North Haugh
St Andrews, Fife, KY16 9SX
United Kingdom

José João Morais

Address: No address known

Copyright

Acknowledgements

The first author wishes to acknowledge Cyril Nicaud and Paulo Varandas for their help in programming some functions of the very first version of this package. He wishes also to acknowledge useful discussions and comments by Cyril Nicaud, Vítor H. Fernandes, Jean-Eric Pin and Jorge Almeida.

The first author also acknowledges support of FCT through CMUP and the FCT and POCTI Project POCTI/32817/MAT/2000 which is funded in cooperation with the European Community Fund FEDER.

The third author acknowledges financial support of FCT and the POCTI program through a scholarship given by Centro de Matemática da Universidade do Porto.

The authors would like to thank Mark Kambites for his contribution in finding bugs and making suggestions for the improvement of this package.

Concerning the maintenance:

The first author was/is (partially) supported by:

the Fundação para a Ciência e a Tecnologia (FCT) project PTDC/MAT/65481/2006;

the *Centro de Matemática da Universidade do Porto* (CMUP), in particular for those periods in which CMUP was funded by

the European Regional Development Fund through the program COMPETE and by the Portuguese Government through the FCT - Fundação para a Ciência e a Tecnologia under the project PEst-C/MAT/UI0144/2011;

The first author wants to acknowledge partial support by CMUP (UID/MAT/00144/2013 and UID/MAT/00144/2019), which is funded by FCT (Portugal) with national (MEC) and European structural funds through the programs FEDER, under the partnership agreement PT2020.

Colophon

This work started in 1998, when the first author was in the LIAFA at the University of Paris 7, in a post-doc. Encouraged by J. E. Pin, he began the implementation in GAP3 of an algorithm obtained some time before to answer a question from the realm of Finite Semigroups proposed by J. Almeida.

The first version of this package on automata was prepared by the first author who gave it the form of a GAP share package. In a second version, prepared by the first and third authors, many functions have been added and the performance of many of the existing ones has been improved. Further important improvements, specially concerning performance, have been achieved when the second author joined the group. The first author is particularly grateful to Max Horn whose help made the release of Versions 1.14 and 1.15 a lot easier.

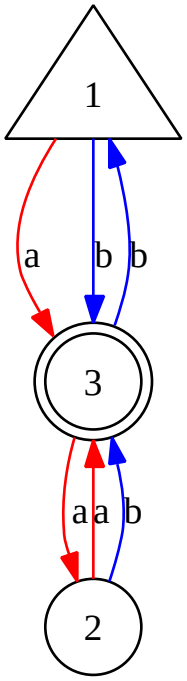
Since Version 1.12, the package is maintained by the first two authors. Bug reports, suggestions and comments are, of course, welcome. Please use our email addresses to this effect.

Contents

Chapter 1

Introduction

In many situations an automaton is conveniently described through a diagram like the following



This diagram describes a (deterministic) automaton with 3 states (the elements of the set $\{1,2,3\}$). The arrow pointing to the state 1 indicates that 1 is the initial state and the two circles around state 3 indicate that 3 is a final or accepting state. The set $\{a,b\}$ is the *alphabet* of the automaton; its elements are called *letters* and are the labels of the edges of the diagram. The words a , ab^2 , b^5a^3b are examples of words recognized by the automaton since they are labels of paths from the initial to the final state.

The set of words recognized by an automaton is called the *language* of the automaton. It is a rational language and may be represented through a rational expression. For instance,

Example

```
(aUb)(a(aUb)Ub(aUb))*
```

is a rational expression representing the language of the above automaton.

Kleene's Theorem states that a language is rational if and only if it is the language of a finite automaton. Both directions of Kleene's Theorem can be proved constructively, and these algorithms, to go from an automaton to a rational expression and *vice-versa*, are implemented in this package.

Of course, one has to pay attention to the size of the output produced. When producing a deterministic automaton equivalent to a given rational expression one can obtain an optimal solution (the minimal automaton) using standard algorithms [AHU74].

When producing a rational expression for the language of an automaton, and taking into account some reasonable measure for the size of a rational expression, to determine a minimal one is apparently computationally difficult. We use here some heuristic methods (to be published elsewhere) which in practice lead to very reasonable results.

The development of this work has benefited from the existence of AMoRE [MMP+95], a package written in C to handle Automata, Monoids and Regular Expressions. In fact, its manual has been very useful and some of the algorithms implemented here are those implemented in AMoRE. In this package, unlike what happened with AMoRE, we do not have to worry about the monoid part in order to make it useful to semigroup theorists, since monoids are already implemented in GAP and we may take advantage of this fact. We just need a function to compute the transition semigroup of an automaton.

The parts of this package that have not so directly to do with automata or rational expressions are put into appendices in this manual. Some words about these appendices follow.

Using the external program Graphviz [DEG+02] to graph visualization, one can visualize automata. This very convenient tool presently works easily under LINUX.

Given a finitely generated subgroup of the free group it is possible to compute a flower automaton and perform Stallings foldings over it in order to obtain an inverse automaton corresponding to the given subgroup.

Chapter 2

Finite Automata

This chapter describes the representations used in this package for finite automata and some functions to determine information about them.

We have to remark that the states of an automaton are always named $1,2,3,\ldots$; the alphabet may be given by the user. By default it is $\{a,b,c,\ldots\}$ (or $\{a_1,a_2,a_3,\ldots\}$ in the case of alphabets with more than 26 letters).

The transition function of an automaton with q states over an alphabet with n letters is represented by a (not necessarily dense) $n \times q$ matrix. Each row of the matrix describes the action of the corresponding letter on the states. In the case of a *deterministic automaton* (DFA) the entries of the matrix are non-negative integers. When all entries of the transition table are positive integers, the automaton is said to be *dense* or *complete*. In the case of a *non deterministic automaton* (NFA) the entries of the matrix may be lists of non-negative integers. *Automata with ε-transitions* are also allowed: the last letter of the alphabet is assumed to be ε and is represented by `@`.

2.1 Automata generation

The way to create an automaton in GAP is the following

2.1.1 Automaton

▷ `Automaton(`*`Type, Size, Alphabet, TransitionTable, Initial, Accepting`*`)` (function)

`Type` may be `"det"`, `"nondet"` or `"epsilon"` according to whether the automaton is deterministic, non deterministic or an automaton with ε-transitions. `Size` is a positive integer representing the number of states of the automaton. `Alphabet` is the number of letters of the alphabet or a list with the letters of the ordered alphabet. `TransitionTable` is the transition matrix. The entries are non-negative integers not greater than the size of the automaton. In the case of non deterministic automata, lists of non-negative integers not greater than the size of the automaton are also allowed. `Initial` and `Accepting` are, respectively, the lists of initial and accepting states.

Example

```
gap> aut:=Automaton("det",4,2,[[3,,3,4],[3,4,0,4]],[1],[4]);
< deterministic automaton on 2 letters with 4 states >
gap> Display(aut);
   |  1  2  3  4
```

```
-----------------
 a |  3     3  4
 b |  3  4     4
Initial state:   [ 1 ]
Accepting state: [ 4 ]
```

The alphabet of the automaton may be specified:

Example

```
gap> aut:=Automaton("det",4,"01",[[3,,3,4],[3,4,0,4]],[1],[4]);
< deterministic automaton on 2 letters with 4 states >
gap> Display(aut);
   |  1  2  3  4
-----------------
 0 |  3     3  4
 1 |  3  4     4
Initial state:   [ 1 ]
Accepting state: [ 4 ]
```

Instead of leaving a hole in the transition matrix, we may write a 0 to mean that no transition is present. Non deterministic automata may be given the same way.

Example

```
gap> ndaut:=Automaton("nondet",4,2,[[3,[1,2],3,0],[3,4,0,[2,3]]],[1],[4]);
< non deterministic automaton on 2 letters with 4 states >
gap> Display(ndaut);
   |  1       2          3       4
----------------------------------------
 a | [ 3 ]   [ 1, 2 ]   [ 3 ]
 b | [ 3 ]   [ 4 ]              [ 2, 3 ]
Initial state:   [ 1 ]
Accepting state: [ 4 ]
```

Also in the same way can be given ε-automata. The letter ε is written @ instead.

Example

```
gap> x:=Automaton("epsilon",3,"01@",[[,[2],[3]],[[1,3],,[1]],[[1],[2],
> [2]]],[2],[2,3]);
< epsilon automaton on 3 letters with 3 states >
gap> Display(x);
   |  1          2       3
------------------------------
 0 |            [ 2 ]   [ 3 ]
 1 | [ 1, 3 ]           [ 1 ]
 @ | [ 1 ]      [ 2 ]   [ 2 ]
Initial state:    [ 2 ]
Accepting states: [ 2, 3 ]
```

Bigger automata are displayed in another form:

Example

```
gap> aut:=Automaton("det",16,2,[[4,0,0,6,3,1,4,8,7,4,3,0,6,1,6,0],
> [3,4,0,0,6,1,0,6,1,6,1,6,6,4,8,7,4,5]],[1],[4]);
< deterministic automaton on 2 letters with 16 states >
```

```
gap> Display(aut);
1    a   4
1    b   3
2    b   4
    ... some more lines
15   a   6
15   b   8
16   b   7
Initial state:   [ 1 ]
Accepting state: [ 4 ]
```

2.1.2 IsAutomaton

▷ `IsAutomaton(`*O*`)` (function)

In the presence of an object *O*, one may want to test whether O is an automaton. This may be done using the function IsAutomaton.

Example

```
gap> x:=Automaton("det",3,2,[ [ 0, 2, 0 ], [ 0, 1, 0 ] ],[ 3 ],[ 2 ]);;
gap> IsAutomaton(x);
true
```

2.1.3 IsDeterministicAutomaton

▷ `IsDeterministicAutomaton(`*aut*`)` (function)

Returns true when aut is a deterministic automaton and false otherwise.

Example

```
gap> x:=Automaton("det",3,2,[ [ 0, 2, 0 ], [ 0, 1, 0 ] ],[ 3 ],[ 2 ]);;
gap> IsDeterministicAutomaton(x);
true
```

2.1.4 IsNonDeterministicAutomaton

▷ `IsNonDeterministicAutomaton(`*aut*`)` (function)

Returns true when aut is a non deterministic automaton and false otherwise.

Example

```
gap> y:=Automaton("nondet",3,2,[[,[1,3],],[,[2,3],[1,3]]],[1,2],[1,3]);;
gap> IsNonDeterministicAutomaton(y);
true
```

2.1.5 IsEpsilonAutomaton

▷ `IsEpsilonAutomaton(`*aut*`)` (function)

Returns true when aut is an ε-automaton and false otherwise.

Example

```
gap> z:=Automaton("epsilon",2,2,[[[1,2],],[[2],[1]]],[1,2],[1,2]);;
gap> IsEpsilonAutomaton(z);
true
```

2.1.6 String

▷ `String(aut)` (function)

The way GAP displays an automaton is quite natural, but when one wants to do small changes, for example using *copy/paste*, the use of the function `String` (possibly followed by `Print`) may be usefull.

Example

```
gap> x:=Automaton("det",3,2,[ [ 0, 2, 0 ], [ 0, 1, 0 ] ],[ 3 ],[ 2 ]);;
gap> String(x);
"Automaton(\"det\",3,\"ab\",[ [ 0, 2, 0 ], [ 0, 1, 0 ] ],[ 3 ],[ 2 ]);;"
gap> Print(String(x));
Automaton("det",3,"ab",[ [ 0, 2, 0 ], [ 0, 1, 0 ] ],[ 3 ],[ 2 ]);;
```

Example

```
gap> z:=Automaton("epsilon",2,2,[[[1,2],],[[2],[1]]],[1,2],[1,2]);;
gap> Print(String(z));
Automaton("epsilon",2,"a@",[ [ [ 1, 2 ], [ ] ], [ [ 2 ], [ 1 ] ] ],[ 1, 2 ],[ \
1, 2 ]);;
```

2.1.7 RandomAutomaton

▷ `RandomAutomaton(Type, Size, Alphabet)` (function)

Given the *Type*, the *Size* (i.e. the number of states) and the *Alphabet* (a positive integer or a list), returns a pseudo random automaton with these parameters.

Example

```
gap> RandomAutomaton("det",5,"ac");
< deterministic automaton on 2 letters with 5 states >
gap> Display(last);
   |  1  2  3  4  5
--------------------
 a |     2  3
 c |  2  3
Initial state:    [ 4 ]
Accepting states: [ 3, 4 ]

gap> RandomAutomaton("nondet",3,["a","b","c"]);
< non deterministic automaton on 3 letters with 3 states >

gap> RandomAutomaton("epsilon",2,"abc");
< epsilon automaton on 4 letters with 2 states >

gap> RandomAutomaton("epsilon",2,2);
< epsilon automaton on 3 letters with 2 states >
gap> Display(last);
```

```
    |  1          2
----------------------
 a | [ 1, 2 ]
 b | [ 2 ]      [ 1 ]
 @ | [ 1, 2 ]
Initial state:    [ 2 ]
Accepting states: [ 1, 2 ]

gap> a:=RandomTransformation(3);;
gap> b:=RandomTransformation(3);;
gap> aut:=RandomAutomaton("det",4,[a,b]);
< deterministic automaton on 2 letters with 4 states >
```

2.2 Automata internals

In this section we describe the functions used to access the internals of an automaton.

2.2.1 AlphabetOfAutomaton

▷ `AlphabetOfAutomaton(`*aut*`)` (function)

Returns the number of symbols in the alphabet of automaton `aut`.

Example

```
gap> aut:=Automaton("det",4,2,[[3,,3,4],[3,4,0,]],[1],[4]);;
gap> AlphabetOfAutomaton(aut);
2
```

2.2.2 AlphabetOfAutomatonAsList

▷ `AlphabetOfAutomatonAsList(`*aut*`)` (function)

Returns the alphabet of automaton `aut` always as a list. Note that when the alphabet of the automaton is given as an integer (meaning the number of symbols) not greater than 26 it returns the list `"abcd..."`. If the alphabet is given by means of an integer greater than 26, the function returns `[ "a1", "a2", "a3", "a4", ... ]`.

Example

```
gap> a:=RandomAutomaton("det",5,"cat");
< deterministic automaton on 3 letters with 5 states >
gap> AlphabetOfAutomaton(a);
3
gap> AlphabetOfAutomatonAsList(a);
"cat"
gap> a:=RandomAutomaton("det",5,20);
< deterministic automaton on 20 letters with 5 states >
gap> AlphabetOfAutomaton(a);
20
gap> AlphabetOfAutomatonAsList(a);
"abcdefghijklmnopqrst"
gap> a:=RandomAutomaton("det",5,30);
```

```
< deterministic automaton on 30 letters with 5 states >
gap> AlphabetOfAutomaton(a);
30
gap> AlphabetOfAutomatonAsList(a);
[ "a1", "a2", "a3", "a4", "a5", "a6", "a7", "a8", "a9", "a10", "a11",
  "a12", "a13", "a14", "a15", "a16", "a17", "a18", "a19", "a20", "a21",
  "a22", "a23", "a24", "a25", "a26", "a27", "a28", "a29", "a30" ]
```

2.2.3 TransitionMatrixOfAutomaton

▷ `TransitionMatrixOfAutomaton(`*aut*`)` (function)

Returns the transition matrix of automaton `aut`.

Example

```
gap> aut:=Automaton("det",4,2,[[3,,3,4],[3,4,0,]],[1],[4]);;
gap> TransitionMatrixOfAutomaton(aut);
[ [ 3, 0, 3, 4 ], [ 3, 4, 0, 0 ] ]
```

2.2.4 InitialStatesOfAutomaton

▷ `InitialStatesOfAutomaton(`*aut*`)` (function)

Returns the initial states of automaton `aut`.

Example

```
gap> aut:=Automaton("det",4,2,[[3,,3,4],[3,4,0,]],[1],[4]);;
gap> InitialStatesOfAutomaton(aut);
[ 1 ]
```

2.2.5 SetInitialStatesOfAutomaton

▷ `SetInitialStatesOfAutomaton(`*aut*`, `*I*`)` (function)

Sets the initial states of automaton `aut`. `I` may be a positive integer or a list of positive integers.

Example

```
gap> aut:=Automaton("det",4,2,[[3,,3,4],[3,4,0,]],[1],[4]);;
gap> SetInitialStatesOfAutomaton(aut,4);
gap> InitialStatesOfAutomaton(aut);
[ 4 ]
gap> SetInitialStatesOfAutomaton(aut,[2,3]);
gap> InitialStatesOfAutomaton(aut);
[ 2, 3 ]
```

2.2.6 FinalStatesOfAutomaton

▷ `FinalStatesOfAutomaton(`*aut*`)` (function)

Returns the final states of automaton `aut`.

Example

```
gap> aut:=Automaton("det",4,2,[[3,,3,4],[3,4,0,]],[1],[4]);;
gap> FinalStatesOfAutomaton(aut);
[ 4 ]
```

2.2.7 SetFinalStatesOfAutomaton

▷ SetFinalStatesOfAutomaton(*aut*, *F*) (function)

Sets the final states of automaton aut. F may be a positive integer or a list of positive integers.

Example

```
gap> aut:=Automaton("det",4,2,[[3,,3,4],[3,4,0,]],[1],[4]);;
gap> FinalStatesOfAutomaton(aut);
[ 4 ]
gap> SetFinalStatesOfAutomaton(aut,2);
gap> FinalStatesOfAutomaton(aut);
[ 2 ]
```

2.2.8 NumberStatesOfAutomaton

▷ NumberStatesOfAutomaton(*aut*) (function)

Returns the number of states of automaton aut.

Example

```
gap> aut:=Automaton("det",4,2,[[3,,3,4],[3,4,0,]],[1],[4]);;
gap> NumberStatesOfAutomaton(aut);
4
```

2.3 Comparison of automata

Although there is no standard way to compare automata it is usefull to be able to do some kind of comparison. Doing so, one can consider sets of automata. We just compare the strings of the automata.

Example

```
gap> x:=Automaton("det",3,2,[ [ 0, 2, 0 ], [ 0, 1, 0 ] ],[ 3 ],[ 2 ]);;
gap> y:=Automaton("det",3,2,[ [ 2, 0, 0 ], [ 1, 3, 0 ] ],[ 3 ],[ 2, 3 ]);;
gap> x=y;
false
gap> Size(Set([y,x,x]));
2
```

2.4 Tests involving automata

This section describes some useful tests involving automata.

2.4.1 IsDenseAutomaton

▷ `IsDenseAutomaton(aut)` (function)

Tests whether a deterministic automaton `aut` is complete. (See also `NullCompletionAutomaton` (2.5.2).)

Example

```
gap> aut:=Automaton("det",4,2,[[3,,3,4],[3,4,0,]],[1],[4]);;
gap> IsDenseAutomaton(aut);
false
```

2.4.2 IsRecognizedByAutomaton

▷ `IsRecognizedByAutomaton(A, w)` (function)

The arguments are: an automaton `A` and a string (i.e. a word) `w` in the alphabet of the automaton. Returns `true` if the word is recognized by the automaton and `false` otherwise.

Example

```
gap> aut:=Automaton("det",3,2,[[1,2,1],[2,1,3]],[1],[2]);;
gap> IsRecognizedByAutomaton(aut,"bbb");
true

gap> aut:=Automaton("det",3,"01",[[1,2,1],[2,1,3]],[1],[2]);;
gap> IsRecognizedByAutomaton(aut,"111");
true
```

2.4.3 IsPermutationAutomaton

▷ `IsPermutationAutomaton(aut)` (function)

The argument is a deterministic automaton. Returns `true` when each letter of the alphabet induces a permutation on the vertices and `false` otherwise.

Example

```
gap> x:=Automaton("det",3,2,[ [ 1, 2, 3 ], [ 1, 2, 3 ] ],[ 1 ],[ 2, 3 ]);;
gap> IsPermutationAutomaton(x);
true
```

2.4.4 IsInverseAutomaton

▷ `IsInverseAutomaton(aut)` (function)

The argument is a deterministic automaton. Returns `true` when each letter of the alphabet induces an injective partial function on the vertices and `false` otherwise.

Example

```
gap> x:=Automaton("det",3,2,[ [ 0, 1, 3 ], [ 0, 1, 2 ] ],[ 2 ],[ 1 ]);;
gap> IsInverseAutomaton(x);
true
```

Frequently an inverse automaton is thought as if the inverse edges (labeled by formal inverses of the letters of the alphabet) were present, although they are usually not explicited. They can be made explicit using the function `AddInverseEdgesToInverseAutomaton`

2.4.5 AddInverseEdgesToInverseAutomaton

▷ AddInverseEdgesToInverseAutomaton(*aut*) (function)

The argument is an inverse automaton over the alphabet $\{a,b,c,\ldots\}$. Returns an automaton with the inverse edges added. (The formal inverse of a letter is represented by the corresponding capital letter.)

Example

```
gap> x:=Automaton("det",3,2,[[ 0, 1, 3 ],[ 0, 1, 2 ]],[ 2 ],[ 1 ]);;Display(x);
   |  1  2  3
--------------
 a |     1  3
 b |     1  2
Initial state:   [ 2 ]
Accepting state: [ 1 ]
gap> AddInverseEdgesToInverseAutomaton(x);Display(x);
   |  1  2  3
--------------
 a |     1  3
 b |     1  2
 A |  2     3
 B |  2  3
Initial state:   [ 2 ]
Accepting state: [ 1 ]
```

2.4.6 IsReversibleAutomaton

▷ IsReversibleAutomaton(*aut*) (function)

The argument is a deterministic automaton. Returns `true` when *aut* is a reversible automaton, i.e. the automaton obtained by reversing all edges and switching the initial and final states (see also `ReversedAutomaton` (2.5.5)) is deterministic. Returns `false` otherwise.

Example

```
gap> x:=Automaton("det",3,2,[ [ 0, 1, 2 ], [ 0, 1, 3 ] ],[ 2 ],[ 2 ]);;
gap> IsReversibleAutomaton(x);
true
```

2.5 Basic operations

2.5.1 CopyAutomaton

▷ CopyAutomaton(*aut*) (function)

Returns a new automaton, which is a copy of automaton *aut*.

2.5.2 NullCompletionAutomaton

▷ NullCompletionAutomaton(*aut*) (function)

aut is a deterministic automaton. If it is complete returns *aut*, otherwise returns the completion (with a null state) of *aut*. Notice that the words recognized by *aut* and its completion are the same.

Example

```
gap> aut:=Automaton("det",4,2,[[3,,3,4],[2,4,4,]],[1],[4]);;
gap> IsDenseAutomaton(aut);
false
gap> y:=NullCompletionAutomaton(aut);;Display(y);
   |  1  2  3  4  5
--------------------
 a |  3  5  3  4  5
 b |  2  4  4  5  5
Initial state:   [ 1 ]
Accepting state: [ 4 ]
```

The state added is a *sink state* i.e. it is a state q which is not initial nor accepting and for all letter a in the alphabet of the automaton, q is the result of the action of a in q. (Notice that reading a word, one does not go out of a sink state.)

2.5.3 ListSinkStatesAut

▷ ListSinkStatesAut(*aut*) (function)

Computes the list of all sink states of the automaton *aut*.

Example

```
gap> x:=Automaton("det",3,2,[ [ 2, 3, 3 ], [ 1, 2, 3 ] ],[ 1 ],[ 2, 3 ]);;
gap> ListSinkStatesAut(x);
[  ]
gap> y:=Automaton("det",3,2,[ [ 2, 3, 3 ], [ 1, 2, 3 ] ],[ 1 ],[ 2 ]);;
gap> ListSinkStatesAut(y);
[ 3 ]
```

2.5.4 RemovedSinkStates

▷ RemovedSinkStates(*aut*) (function)

Removes all sink states of the automaton *aut*.

Example

```
gap> y:=Automaton("det",3,2,[[ 2, 3, 3 ],[ 1, 2, 3 ]],[ 1 ],[ 2 ]);;Display(y);
   |  1  2  3
--------------
 a |  2  3  3
 b |  1  2  3
Initial state:   [ 1 ]
Accepting state: [ 2 ]
gap> x := RemovedSinkStates(y);Display(x);
< deterministic automaton on 2 letters with 2 states >
   |  1  2
-----------
 a |  2
 b |  1  2
Initial state:   [ 1 ]
Accepting state: [ 2 ]
```

2.5.5 ReversedAutomaton

▷ ReversedAutomaton(*aut*) (function)

Inverts the arrows of the automaton *aut*.

Example

```
gap> y:=Automaton("det",3,2,[ [ 2, 3, 3 ], [ 1, 2, 3 ] ],[ 1 ],[ 2 ]);;
gap> z:=ReversedAutomaton(y);;Display(z);
   |  1       2       3
------------------------------
 a |         [ 1 ]   [ 2, 3 ]
 b | [ 1 ]   [ 2 ]   [ 3 ]
Initial state:   [ 2 ]
Accepting state: [ 1 ]
```

2.5.6 PermutedAutomaton

▷ PermutedAutomaton(*aut*, *p*) (function)

Given an automaton *aut* and a list *p* representing a permutation of the states, outputs the equivalent permuted automaton.

Example

```
gap> y:=Automaton("det",4,2,[[2,3,4,2],[0,0,0,1]],[1],[3]);;Display(y);
   |  1  2  3  4
-----------------
 a |  2  3  4  2
 b |           1
Initial state:   [ 1 ]
Accepting state: [ 3 ]
gap> Display(PermutedAutomaton(y, [3,2,4,1]));
   |  1  2  3  4
-----------------
 a |  2  4  2  1
 b |  3
Initial state:   [ 3 ]
Accepting state: [ 4 ]
```

2.5.7 ListPermutedAutomata

▷ ListPermutedAutomata(*aut*) (function)

Given an automaton *aut*, returns a list of automata with permuted states

Example

```
gap> x:=Automaton("det",3,2,[ [ 0, 2, 3 ], [ 1, 2, 3 ] ],[ 1 ],[ 2, 3 ]);;
gap> ListPermutedAutomata(x);
[ < deterministic automaton on 2 letters with 3 states >,
  < deterministic automaton on 2 letters with 3 states >,
  < deterministic automaton on 2 letters with 3 states >,
  < deterministic automaton on 2 letters with 3 states >,
  < deterministic automaton on 2 letters with 3 states >,
  < deterministic automaton on 2 letters with 3 states > ]
```

2.5.8 NormalizedAutomaton

▷ NormalizedAutomaton(*A*) (function)

Produces an equivalent automaton but in which the initial state is numbered 1 and the accepting states have the greatest numbers.

Example

```
gap> x:=Automaton("det",3,2,[[ 1, 2, 0 ],[ 0, 1, 2 ]],[2],[1, 2]);;Display(x);
   |  1  2  3
--------------
 a |  1  2
 b |     1  2
Initial state:    [ 2 ]
Accepting states: [ 1, 2 ]
gap> Display(NormalizedAutomaton(x));
   |  1  2  3
--------------
 a |  1     3
 b |  3  1
Initial state:    [ 1 ]
Accepting states: [ 3, 1 ]
```

2.5.9 UnionAutomata

▷ UnionAutomata(*A, B*) (function)

Produces the disjoint union of the deterministic or non deterministic automata A and B. The output is a non-deterministic automaton.

Example

```
gap> x:=Automaton("det",3,2,[ [ 1, 2, 0 ], [ 0, 1, 2 ] ],[ 2 ],[ 1, 2 ]);;
gap> y:=Automaton("det",3,2,[ [ 0, 1, 3 ], [ 0, 0, 0 ] ],[ 1 ],[ 1, 2, 3 ]);;
gap> UnionAutomata(x,y);
< non deterministic automaton on 2 letters with 6 states >
gap> Display(last);
   |  1       2       3       4    5       6
-----------------------------------------------
 a | [ 1 ]   [ 2 ]                [ 4 ]   [ 6 ]
 b |         [ 1 ]   [ 2 ]
Initial states:   [ 2, 4 ]
Accepting states: [ 1, 2, 4, 5, 6 ]
```

2.5.10 ProductAutomaton

▷ ProductAutomaton(*A1, A2*) (function)

The arguments must be deterministic automata. Returns the product of *A1* and *A2*.
Note: $(p,q) -> (p-1)m+q$ is a bijection from $\{1,\ldots,n\}\times\{1,\ldots,m\}$ to $\{1,\ldots,mn\}$.

Example

```
gap> x := Automaton("det",3,"ab",[ [ 0, 1, 2 ], [ 1, 3, 0 ] ],[ 1 ],[ 1, 2 ]);;
gap> Display(x);
   |  1  2  3
```

```
--------------
 a |     1  2
 b |  1  3
Initial state:    [ 1 ]
Accepting states: [ 1, 2 ]
gap> y := Automaton("det",3,"ab",[ [ 0, 1, 2 ], [ 0, 2, 0 ] ],[ 2 ],[ 1, 2 ]);;
gap> Display(y);
   |  1  2  3
--------------
 a |     1  2
 b |     2
Initial state:    [ 2 ]
Accepting states: [ 1, 2 ]
gap> z:=ProductAutomaton(x, y);;Display(z);
   |  1  2  3  4  5  6  7  8  9
--------------------------------
 a |              1  2     4  5
 b |     2        8
Initial state:    [ 2 ]
Accepting states: [ 1, 2, 4, 5 ]
```

2.5.11 ProductOfLanguages

▷ `ProductOfLanguages(`*A1, A2*`)` (function)

Given two regular languages (as automata or rational expressions), returns an automaton that recognizes the concatenation of the given languages, that is, the set of words *uv* such that *u* belongs to the first language and *v* belongs to the second language.

Example

```
gap> a1:=ListOfWordsToAutomaton("ab",["aa","bb"]);
< deterministic automaton on 2 letters with 5 states >
gap> a2:=ListOfWordsToAutomaton("ab",["a","b"]);
< deterministic automaton on 2 letters with 3 states >
gap> ProductOfLanguages(a1,a2);
< deterministic automaton on 2 letters with 5 states >
gap> FAtoRatExp(last);
(bbUaa)(aUb)
```

2.6 Links with Semigroups

Each letter of the alphabet of an automaton induces a partial transformation in its set of states. The semigroup generated by these transformations is called the *transition semigroup* of the automaton.

2.6.1 TransitionSemigroup

▷ `TransitionSemigroup(`*aut*`)` (function)

Returns the transition semigroup of the deterministic automaton *aut*.

Example

```
gap> aut := Automaton("det",10,2,[[7,5,7,5,4,9,10,9,10,9],
> [8,6,8,9,9,1,3,1,9,9]],[2],[6,7,8,9,10]);;
gap> s := TransitionSemigroup(aut);;
gap> Size(s);
30
```

The transition semigroup of the minimal automaton recognizing a language is the *syntactic semigroup* of that language.

2.6.2 SyntacticSemigroupAut

▷ SyntacticSemigroupAut(*aut*) (function)

Returns the syntactic semigroup of the deterministic automaton *aut* (i.e. the transition semigroup of the equivalent minimal automaton) when it is non empty and returns fail otherwise.

Example

```
gap> x:=Automaton("det",3,2,[ [ 1, 2, 0 ], [ 0, 1, 2 ] ],[ 2 ],[ 1, 2 ]);;
gap> S:=SyntacticSemigroupAut(x);;
gap> Size(S);
3
```

2.6.3 SyntacticSemigroupLang

▷ SyntacticSemigroupLang(*rat*) (function)

Returns the syntactic semigroup of the language given by the rational expression *rat*.

Example

```
gap> rat := RationalExpression("a*ba*ba*(@Ub)");;
gap> S:=SyntacticSemigroupLang(rat);;
gap> Size(S);
7
```

Chapter 3

Rational languages

Rational languages are conveniently represented through rational expressions. These are finite expressions involving letters of the alphabet; `concatenation`, corresponding to the *product*; the symbol `U`, corresponding to the *union*; and the symbol `*`, corresponding to the Kleene's star.

3.1 Rational Expressions

The expressions `@` and `"empty\_set"` are used to represent the empty word and the empty set respectively.

3.1.1 RationalExpression

▷ `RationalExpression(`*`expr[, alph]`*`)` (function)

A rational expression can be created using the function `RationalExpression`. *`expr`* is a string representing the desired expression literally and *`alph`* (may or may not be present) is the alphabet of the expression. Of course *`alph`* must not contain the symbols '@', '(', ')', '*' nor 'U'. When *`alph`* is not present, the alphabet of the rational expression is the set of symbols (other than '"', etc...) occurring in the expression. (The alphabet is then ordered with the alphabetical order.)

Example

```
gap> RationalExpression("abUc");
abUc
gap> RationalExpression("ab*Uc");
ab*Uc
gap> RationalExpression("001U1*");
001U1*
gap> RationalExpression("001U1*","012");
001U1*
```

3.1.2 RatExpOnnLetters

▷ `RatExpOnnLetters(`*`n, operation, list`*`)` (function)

This is another way to construct a rational expression over an alphabet. The user may specify the alphabet or just give the number n of letters (in this case the alphabet $\{a,b,c,\ldots\}$ is considered).

operation is the name of an operation, the possibilities being: `product`, `union` or `star`. *list* is a list of rational expressions, a rational expression in the case of "star", or a list consisting of an integer when the rational expression is a single letter. The empty list `[ ]` and `empty\_set` are other possibilities for `list`. An example follows

Example

```
gap> RatExpOnnLetters(2,"star",RatExpOnnLetters(2,"product",
> [RatExpOnnLetters(2,[],[1]),RatExpOnnLetters(2,"union",
> [RatExpOnnLetters(2,[],[1]),RatExpOnnLetters(2,[],[2])])]));
(a(aUb))*
```

The empty word and the empty set are the rational expressions:

Example

```
gap> RatExpOnnLetters(2,[],[]);
@
gap> RatExpOnnLetters(2,[],"empty_set");
empty_set
```

3.1.3 RandomRatExp

▷ `RandomRatExp(`*arg*`)` (function)

Given the number of symbols of the alphabet and (possibly) a factor *m* which is intended to increase the randomality of the expression, returns a pseudo random rational expression over that alphabet.

Example

```
gap> RandomRatExp(2);
b*(@Ua)
gap> RandomRatExp("01");
empty_set
gap> RandomRatExp("01");
(0U1)*
gap> RandomRatExp("01",7);
0*1(@U0U1)
```

3.1.4 SizeRatExp

▷ `SizeRatExp(`*r*`)` (function)

Returns the size, i.e. the number of symbols of the alphabet, of the rational expression *r*.

Example

```
gap> a:=RationalExpression("0*1(@U0U1)");
0*1(@U0U1)
gap> SizeRatExp(a);
5
```

3.1.5 IsRationalExpression

▷ `IsRationalExpression(`*r*`)` (function)

Tests whether an object is a rational expression.

Example

```
gap> r := RationalExpression("1(0U1)U@");
1(0U1)U@
gap> IsRatExpOnnLettersObj(r) and IsRationalExpressionRep(r);
true
gap> IsRationalExpression(RandomRatExp("01"));
true
```

3.1.6 AlphabetOfRatExp

▷ `AlphabetOfRatExp(`*R*`)` (function)

Returns the number of symbols in the alphabet of the rational expression `R`.

Example

```
gap> r:=RandomRatExp(2);
a*(ba*U@)
gap> AlphabetOfRatExp(r);
2
gap> r:=RandomRatExp("01");
1*(01*U@)
gap> AlphabetOfRatExp(r);
2
gap> a:=RandomTransformation(3);;
gap> b:=RandomTransformation(3);;
gap> r:=RandomRatExp([a,b]);
(Transformation( [ 1, 1, 3 ] )UTransformation( [ 1, 1, 2 ] ))*
gap> AlphabetOfRatExp(r);
2
```

3.1.7 AlphabetOfRatExpAsList

▷ `AlphabetOfRatExpAsList(`*R*`)` (function)

Returns the alphabet of the rational expression `R` always as a list. If the alphabet of the rational expression is given by means of an integer less than 27 it returns the list `"abcd...."`, otherwise returns `[ "a1", "a2", "a3", "a4", ... ]`.

Example

```
gap> r:=RandomRatExp(2);
(aUb)((aUb)(bU@)U@)U@
gap> AlphabetOfRatExpAsList(r);
"ab"
gap> r:=RandomRatExp("01");
1*(0U@)
gap> AlphabetOfRatExpAsList(r);
"01"
gap> r:=RandomRatExp(30);;
gap> AlphabetOfRatExpAsList(r);
[ "a1", "a2", "a3", "a4", "a5", "a6", "a7", "a8", "a9", "a10", "a11",
"a12", "a13", "a14", "a15", "a16", "a17", "a18", "a19", "a20", "a21",
"a22", "a23", "a24", "a25", "a26", "a27", "a28", "a29", "a30" ]
```

3.1.8 CopyRatExp

▷ `CopyRatExp(R)` (function)

Returns a new rational expression, which is a copy of `R`.

Example

```
gap> r:=RandomRatExp(2);
a*(bU@)
gap> CopyRatExp(r);
a*(bU@)
```

3.2 Comparison of rational expressions

The way two rational expressions `r1` and `r2` are compared through the $<$ operator is the following: the empty set is lesser than everything else; if r1 and r2 are letters, then the lesser is taken from the order in the alphabet; if r1 is a letter an r2 a product, union or star, then r1 is lesser than r2; a star expression is considered to be lesser than a product or union expression and a product lesser than a union; to compare two star expressions we compare the expressions under the stars; to compare two product or union expressions we compare the subexpressions of each expression from left to right;

3.3 Operations with rational languages

Only operations with rational languages over the same alphabet are allowed.

We may compute expressions for the `product`, `union` and `star` (i.e., submonoid generated by) of rational sets. In some cases, simpler expressions representing the same set are returned. Note that that two simplifications are always made, namely, $r \cup$"$empty_set$" $= r$ and $r\varepsilon = r$. Of course, these operations may be used to construct more complex expressions. For rational expressions we have the functions `UnionRatExp`, `ProductRatExp`, `StarRatExp`, that return respectively rational expressions for the *union* and *product* of the languages given by the rational expressions `r` and `s` and the `star` of the language given by the rational expression `r`.

3.3.1 UnionRatExp

▷ `UnionRatExp(r, s)` (function)

3.3.2 ProductRatExp

▷ `ProductRatExp(r, s)` (function)

3.3.3 StarRatExp

▷ `StarRatExp(r)` (function)

The expression `(a(aUb))*` may be produced in the following way

Example

```
gap> r1 := RatExpOnnLetters(2,[],[1]);
a
gap> r2 := RatExpOnnLetters(2,[],[2]);
b
gap> r3 := UnionRatExp(r1,r2);
aUb
gap> r4 := ProductRatExp(r1,r3);
a(aUb)
gap> r5 := StarRatExp(r4);
(a(aUb))*
```

Chapter 4

Automata *versus* rational expressions

A remarkable theorem due to Kleene shows that a language is recognized by a finite automaton precisely when it may be given by means of a rational expression, i.e. is a rational language.

An automaton and a rational expression are said to be *equivalent* when both represent the same language. In this chapter we describe functions to go from automata to equivalent rational expressions and *vice-versa*.

4.1 From automata to rational expressions

4.1.1 AutomatonToRatExp

▷ `AutomatonToRatExp (A)` (function)
▷ `AutToRatExp(A)` (function)
▷ `FAtoRatExp(A)` (function)

From a finite automaton, `FAtoRatExp`, `AutToRatExp` and `AutomatonToRatExp`, which are synonyms, compute one equivalent rational expression, using the state elimination algorithm.

Example

```
gap> x:=Automaton("det",4,2,[[ 0, 1, 2, 3 ],[ 0, 1, 2, 3 ]],[ 3 ],[ 1, 3, 4 ]);;
gap> FAtoRatExp(x);
(aUb)(aUb)U@
gap> aut:=Automaton("det",4,"xy",[[ 0, 1, 2, 3 ],[ 0, 1, 2, 3 ]],[ 3 ],[ 1, 3, 4 ]);;
gap> FAtoRatExp(aut);
(xUy)(xUy)U@
```

4.2 From rational expression to automata

4.2.1 RatExpToNDAut

▷ `RatExpToNDAut(R)` (function)

Given a rational expression R, computes the equivalent NFA

Example

```
gap> r:=RationalExpression("aUab");
aUab
```

```
gap> Display(RatExpToNDAut(r));
   |  1    2       3    4
--------------------------------
 a |                   [ 1, 2 ]
 b |      [ 3 ]
Initial state:    [ 4 ]
Accepting states: [ 1, 3 ]
gap> r:=RationalExpression("xUxy");
xUxy
gap> Display(RatExpToNDAut(r));
   |  1    2       3    4
--------------------------------
 x |                   [ 1, 2 ]
 y |      [ 3 ]
Initial state:    [ 4 ]
Accepting states: [ 1, 3 ]
```

4.2.2 RatExpToAutomaton

▷ `RatExpToAutomaton(`*R*`)` (function)
▷ `RatExpToAut(`*R*`)` (function)

Given a rational expression *R*, these functions, which are synonyms, use `RatExpToNDAut` (4.2.1)) to compute the equivalent NFA and then returns the equivalent minimal DFA

Example

```
gap> r:=RationalExpression("@U(aUb)(aUb)");
@U(aUb)(aUb)
gap> Display(RatExpToAut(r));
   |  1  2  3  4
-----------------
 a |  3  1  3  2
 b |  3  1  3  2
Initial state:    [ 4 ]
Accepting states: [ 1, 4 ]
gap> r:=RationalExpression("@U(0U1)(0U1)");
@U(0U1)(0U1)
gap> Display(RatExpToAut(r));
   |  1  2  3  4
-----------------
 0 |  3  1  3  2
 1 |  3  1  3  2
Initial state:    [ 4 ]
Accepting states: [ 1, 4 ]
```

4.3 Some tests on automata

This section describes functions that perform tests on automata, rational expressions and their accepted languages.

4.3.1 IsEmptyLang

▷ IsEmptyLang(*R*) (function)

This function tests if the language given through a rational expression or an automaton *R* is the empty language.

Example

```
gap> r := RationalExpression("empty_set");
empty_set
gap> IsEmptyLang(r);
true
gap> r := RationalExpression("aUb");
aUb
gap> IsEmptyLang(r);
false
```

4.3.2 IsFullLang

▷ IsFullLang(*R*) (function)

This function tests if the language given through a rational expression or an automaton *R* represents the Kleene closure of the alphabet of *R* .

Example

```
gap> r:=RationalExpression("aUb");
aUb
gap> IsFullLang(r);
false
gap> r:=RationalExpression("(aUb)*");
(aUb)*
gap> IsFullLang(r);
true
```

4.3.3 AreEqualLang

▷ AreEqualLang(*A1*, *A2*) (function)
▷ AreEquivAut(*A1*, *A2*) (function)

These functions, which are synonyms, test if the automata or rational expressions *A1* and *A2* are equivalent, i.e. represent the same language. This is performed by checking that the corresponding minimal automata are isomorphic. Note hat this cannot happen if the alphabets are different.

Example

```
gap> x := Automaton("det",4,"ab",[ [ 0, 1, 3, 4 ], [ 0, 1, 2, 0 ] ],[ 2 ],[ 1, 2, 3, 4 ] );;
gap> y := Automaton("det",4,"ab",[ [ 1, 3, 4, 0 ], [ 0, 3, 4, 0 ] ],[ 3 ],[ 1, 3, 4 ]);;
gap> z := Automaton("det",4,"ab",[ [ 0, 4, 0, 0 ], [ 1, 3, 4, 0 ] ],[ 2 ],[ 1, 3, 4 ]);;
gap> AreEquivAut(x, y);
true
gap> AreEquivAut(x, z);
false
gap> a:=RationalExpression("(aUb)*ab*");
(aUb)*ab*
```

```
gap> b:=RationalExpression("(aUb)*");
(aUb)*
gap> AreEqualLang(a, b);
false
gap> a:=RationalExpression("(bUa)*");
(bUa)*
gap> AreEqualLang(a, b);
true
gap> x:=RationalExpression("(1U0)*");
(1U0)*
gap> AreEqualLang(a, x);
The given languages are not over the same alphabet
false
```

4.3.4 IsContainedLang

▷ IsContainedLang(*R1, R2*) (function)

Tests if the language represented (through an automaton or a rational expression) by *R1* is contained in the language represented (through an automaton or a rational expression) by *R2* .

Example

```
gap> r:=RationalExpression("aUab");
aUab
gap> s:=RationalExpression("a","ab");
a
gap> IsContainedLang(s,r);
true
gap> IsContainedLang(r,s);
false
```

4.3.5 AreDisjointLang

▷ AreDisjointLang(*R1, R2*) (function)

Tests if the languages represented (through automata or a rational expressions) by *R1* and *R2* are disjoint.

Example

```
gap> r:=RationalExpression("aUab");;
gap> s:=RationalExpression("a","ab");;
gap> AreDisjointLang(r,s);
false
```

Chapter 5

Some functions involving automata

This chapter describes some functions involving automata. It starts with functions to obtain equivalent automata of other type. Then the minimalization is considered.

5.1 From one type to another

Recall that two automata are said to be equivalent when they recognize the same language. Next we have functions which have as input automata of one type and as output equivalent automata of another type.

5.1.1 EpsilonToNFA

▷ `EpsilonToNFA(A)` (function)

A is an automaton with ε-transitions. Returns a NFA recognizing the same language.

Example

```
gap> x := Automaton("epsilon",3,"ab@",[ [ [ 3 ], [ 1 ], [ 1, 2 ] ], [ [ ], [ ], [ 1, 3 ] ], [ [ 1
gap> Display(x);
   |  1          2       3
---------------------------------
 a | [ 3 ]      [ 1 ]   [ 1, 2 ]
 b |                    [ 1, 3 ]
 @ | [ 1, 3 ]   [ 1 ]   [ 3 ]
Initial states:   [ 1, 2, 3 ]
Accepting states: [ 1, 3 ]
gap> Display(EpsilonToNFA(x));
   |  1       2          3
-----------------------------------
 a | [ 3 ]   [ 1, 3 ]   [ 1 .. 3 ]
 b |                    [ 1, 3 .. 3 ]
Initial states:   [ 1 .. 3 ]
Accepting states: [ 1, 2, 3 ]
```

5.1.2 EpsilonToNFASet

▷ `EpsilonToNFASet(A)` (function)

A is an automaton with ε-transitions. Returns a NFA recognizing the same language. This function differs from `EpsilonToNFA` (5.1.1) in that it is faster for smaller automata, or automata with few epsilon transitions, but slower in the really hard cases.

5.1.3 EpsilonCompactedAut

▷ `EpsilonCompactedAut(A)` (function)

A is an automaton with ε-transitions. Returns an εNFA with each strongly-connected component of the epsilon-transitions digraph of A identified with a single state and recognizing the same language.

Example

```
gap> x := Automaton("epsilon",3,"ab@",[ [ [ ], [ 1 ], [ 1 ] ], [ [ 2 ], [ ], [ ] ], [ [ 2 ], [ 1,
gap> Display(x);
   |  1       2            3
-------------------------------------
 a |         [ 1 ]         [ 1 ]
 b | [ 2 ]
 @ | [ 2 ]   [ 1, 2, 3 ]   [ 1, 3 ]
Initial state:    [ 3 ]
Accepting states: [ 2, 3 ]
gap> Display(EpsilonCompactedAut(x));
   |  1
-----------
 a | [ 1 ]
 b | [ 1 ]
 @ |
Initial state:   [ 1 ]
Accepting state: [ 1 ]
```

5.1.4 ReducedNFA

▷ `ReducedNFA(A)` (function)

A is a non deterministic automaton (without ε-transitions). Returns an NFA accepting the same language as its input but with possibly fewer states (it quotients out by the smallest right-invariant partition of the states). A paper describing the algorithm is in preparation.

Example

```
gap> x := Automaton("nondet",5,"ab",[ [ [ 2, 5 ], [ 2, 5 ], [ 4 ], [ 5 ], [ 3 ] ], [ [ 1 , 2 ], [
gap> Display(x);
   |  1          2               3               4          5
-------------------------------------------------------------------------
 a | [ 2, 5 ]   [ 2, 5 ]         [ 4 ]           [ 5 ]      [ 3 ]
 b | [ 1, 2 ]   [ 1, 2, 3, 4 ]   [ 1, 2, 3, 5 ]   [ 2, 5 ]   [ 2, 3 ]
Initial state:    [ 1 ]
Accepting states: [ 2, 3, 5 ]
gap> Display(ReducedNFA(x));
   |  1          2          3               4               5
```

```
-------------------------------------------------------------------------
 a | [ 3 ]      [ 1 ]      [ 2 ]           [ 1, 4 ]         [ 1, 4 ]
 b | [ 3, 4 ]   [ 1, 4 ]   [ 1, 3, 4, 5 ]   [ 2, 3, 4, 5 ]   [ 4, 5 ]
Initial state:    [ 5 ]
Accepting states: [ 1, 3, 4 ]
```

5.1.5 NFAtoDFA

▷ `NFAtoDFA(A)` (function)

Given an NFA, these synonym functions, compute the equivalent DFA, using the powerset construction, according to the algorithm presented in the report of the AMoRE [MMP+95] program. The returned automaton is dense deterministic

Example

```
gap> x:=RandomAutomaton("nondet",3,2);;Display(x);
   |  1       2    3
---------------------------
 a | [ 2 ]        [ 1, 3 ]
 b |              [ 2, 3 ]
Initial states:   [ 1, 3 ]
Accepting states: [ 1, 2 ]
gap> Display(NFAtoDFA(x));
   |  1  2  3
--------------
 a |  2  2  1
 b |  3  3  3
Initial state:    [ 1 ]
Accepting states: [ 1, 2, 3 ]
```

5.1.6 FuseSymbolsAut

▷ `FuseSymbolsAut(A, s1, s2)` (function)

Given an automaton *A* and integers *s1* and *s2* which, returns an NFA obtained by replacing all transitions through *s2* by transitions through *s1*.

Example

```
gap> x:=RandomAutomaton("det",3,2);;Display(x);
   |  1  2  3
--------------
 a |  2  3
 b |     1
Initial state:    [ 3 ]
Accepting states: [ 1, 2, 3 ]
gap> Display(FuseSymbolsAut(x,1,2));
   |  1       2          3
---------------------------
 a | [ 2 ]   [ 1, 3 ]
Initial state:    [ 3 ]
Accepting states: [ 1, 2, 3 ]
```

5.2 Minimalization of an automaton

The algorithm used to minimalize a dense deterministic automaton (i.e., to compute a dense minimal automaton recognizing the same language) is based on an algorithm due to Hopcroft (see [AHU74]). It is well known (see [HU69]) that it suffices to reduce the automaton given and remove the inaccessible states. Again, the documentation for the computer program AMoRE [MMP+95] has been very useful.

5.2.1 UsefulAutomaton

▷ `UsefulAutomaton(A)` (function)

Given an automaton `A` (deterministic or not), outputs a dense DFA `B` whose states are all reachable and such that `A` and `B` are equivalent.

Example

```
gap> x:=RandomAutomaton("det",4,2);;Display(x);
   |  1  2  3  4
-----------------
 a |     3  4
 b |  1  4
Initial state:    [ 3 ]
Accepting states: [ 2, 3, 4 ]
gap> Display(UsefulAutomaton(x));
   |  1  2  3
--------------
 a |  2  3  3
 b |  3  3  3
Initial state:    [ 1 ]
Accepting states: [ 1, 2 ]
```

5.2.2 MinimalizedAut

▷ `MinimalizedAut(A)` (function)

Returns the minimal automaton equivalent to `A`.

Example

```
gap> x:=RandomAutomaton("det",4,2);;Display(x);
   |  1  2  3  4
-----------------
 a |     3  4
 b |  1  2  3
Initial state:    [ 4 ]
Accepting states: [ 2, 3, 4 ]
gap> Display(MinimalizedAut(x));
   |  1  2
-----------
 a |  2  2
 b |  2  2
Initial state:   [ 1 ]
Accepting state: [ 1 ]
```

5.2.3 MinimalAutomaton

▷ `MinimalAutomaton(A)` (attribute)

Returns the minimal automaton equivalent to `A`, but stores it so that future computations of this automaton just return the stored automaton.

Example

```
gap> x:=RandomAutomaton("det",4,2);;Display(x);
   |  1  2  3  4
-----------------
 a |     2  4
 b |     3  4
Initial state:    [ 4 ]
Accepting states: [ 1, 2, 3 ]
gap> Display(MinimalAutomaton(x));
   |  1
--------
 a |  1
 b |  1
Initial state:   [ 1 ]
Accepting state:
```

5.2.4 AccessibleStates

▷ `AccessibleStates(aut[, p])` (function)

Computes the list of states of the automaton `aut` which are accessible from state `p`. When `p` is not given, returns the states which are accessible from any initial state.

Example

```
gap> x:=RandomAutomaton("det",4,2);;Display(x);
   |  1  2  3  4
-----------------
 a |     1  2  4
 b |     2  4
Initial state:    [ 2 ]
Accepting states: [ 1, 2, 3 ]
gap> AccessibleStates(x,3);
[ 1, 2, 3, 4 ]
```

5.2.5 AccessibleAutomaton

▷ `AccessibleAutomaton(A)` (function)

If `A` is a deterministic automaton, not necessarily dense, an equivalent dense deterministic accessible automaton is returned. (The function `UsefulAutomaton` is called.)

If `A` is not deterministic with a single initial state, an equivalent accessible automaton is returned.

Example

```
gap> x:=RandomAutomaton("det",4,2);;Display(x);
   |  1  2  3  4
-----------------
```

```
 a |  1  3
 b |  1  3  4
Initial state:    [ 2 ]
Accepting states: [ 3, 4 ]
gap> Display(AccessibleAutomaton(x));
   |  1  2  3  4
-----------------
 a |  2  4  4  4
 b |  2  3  4  4
Initial state:    [ 1 ]
Accepting states: [ 2, 3 ]
```

5.2.6 IntersectionLanguage

▷ `IntersectionLanguage(A1, A2)` (function)
▷ `IntersectionAutomaton(A1, A2)` (function)

Computes an automaton that recognizes the intersection of the languages given (through automata or rational expressions by) *A1* and *A2*. When the arguments are deterministic automata, is the same as ProductAutomaton, but works for all kinds of automata. Note that the language of the product of two automata is precisely the intersection of the languages of the automata.

Example

```
gap> x:=RandomAutomaton("det",3,2);;Display(x);
   |  1  2  3
--------------
 a |  2  3
 b |     1
Initial state:    [ 3 ]
Accepting states: [ 1, 2, 3 ]
gap> y:=RandomAutomaton("det",3,2);;Display(y);
   |  1  2  3
--------------
 a |     1
 b |  1  3
Initial state:    [ 3 ]
Accepting states: [ 1, 3 ]
gap> Display(IntersectionLanguage(x,y));
   |  1  2
-----------
 a |  2  2
 b |  2  2
Initial state:   [ 1 ]
Accepting state: [ 1 ]
```

5.2.7 AutomatonAllPairsPaths

▷ `AutomatonAllPairsPaths(A)` (function)

Given an automaton *A*, with n states, outputs a n x n matrix P, such that P[i][j] is the list of simple paths from state i to state j in *A*.

Example

```
gap> a:=RandomAutomaton("det",3,2);
< deterministic automaton on 2 letters with 3 states >
gap> AutomatonAllPairsPaths(a);
[ [ [ [ 1, 1 ] ], [  ], [  ] ], [ [ [ 2, 1 ] ], [ [ 2, 2 ] ], [  ] ],
  [ [ [ 3, 2, 1 ] ], [ [ 3, 2 ] ], [ [ 3, 3 ] ] ] ]
gap> Display(a);
   |  1  2  3
--------------
 a |     1  2
 b |  1  2  3
Initial state:    [ 3 ]
Accepting states: [ 1, 2 ]
```

Chapter 6

Finite regular languages

This chapter describes some functions to deal with finite regular languages.

6.1 Dealing with finite regular languages

6.1.1 IsFiniteRegularLanguage

▷ `IsFiniteRegularLanguage(`*L*`)` (function)

L is an automaton or a rational expression. This function tests whether its argument represents a finite language or not.

Example

```
gap> r:=RationalExpression("b*(aU@)");;
gap> IsFiniteRegularLanguage(last);
false
gap> r:=RationalExpression("aUbU@");;
gap> IsFiniteRegularLanguage(last);
true
```

6.1.2 FiniteRegularLanguageToListOfWords

▷ `FiniteRegularLanguageToListOfWords(`*L*`)` (function)

L is an automaton or a rational expression. This function outputs the recognized language as a list of words.

Example

```
gap> r:=RationalExpression("aaUx(aUb)");
aaUx(aUb)
gap>  FiniteRegularLanguageToListOfWords(r);
[ "aa", "xa", "xb" ]
```

6.1.3 ListOfWordsToAutomaton

▷ `ListOfWordsToAutomaton(`*alph*`, `*L*`)` (function)

Given an alphabet *alph* (a list) and a list of words *L* (a list of lists), outputs an automaton that recognizes the given list of words.

Example

```
gap> ListOfWordsToAutomaton("ab",["aaa","bba",""]);
< deterministic automaton on 2 letters with 6 states >
gap> FAtoRatExp(last);
(bbUaa)aU@
```

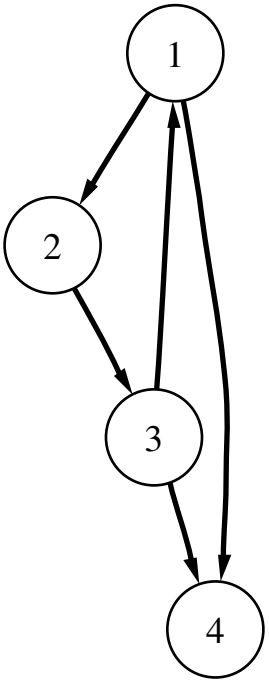
Appendix A

Directed graphs

Automata are frequently described through directed labeled graphs. This appendix on directed graphs (digraphs) is devoted to some functions designed with the purpose of being used as auxiliary functions to deal with automata, but may have independent interest.

A.1 Directed graphs

A directed graph with n vertices is represented by an adjacency list. For example, the list `G = [[2,4],[3],[1,4],[]]` represents a directed graph with `4 (= Length(G))` vertices; the sublist in position `i` is the list of endpoints of the edges beginning in vertex i.



A.1.1 RandomDiGraph

▷ `RandomDiGraph(n)` (function)

Produces a pseudo random digraph with n vertices

Example

```
gap> RandomDiGraph(4);
[ [  ], [ 1, 2, 4 ], [  ], [  ] ]
```

A.1.2 VertexInDegree

▷ `VertexInDegree(DG, v)` (function)

Computes the in degree of the vertex v of the directed graph DG

Example

```
gap> G:= [ [ 1 ], [ 1, 2, 4 ], [  ], [ 1, 2, 3 ] ];;
gap> VertexInDegree(G,2);
2
```

A.1.3 VertexOutDegree

▷ `VertexOutDegree(DG, v)` (function)

Computes the out degree of the vertex v of the directed graph DG

Example

```
gap> G:=[ [ 1 ], [ 1, 2, 4 ], [  ], [ 1, 2, 3 ] ];;
gap> VertexOutDegree(G,2);
3
```

A.1.4 AutoVertexDegree

▷ `AutoVertexDegree(DG, v)` (function)

Computes the degree of the vertex v of the directed graph DG

Example

```
gap> G:=[ [ 1 ], [ 1, 2, 4 ], [  ], [ 1, 2, 3 ] ];;
gap> AutoVertexDegree(G,2);
5
```

A.1.5 ReversedGraph

▷ `ReversedGraph(G)` (function)

Computes the reversed graph of the directed graph G. It is the graph obtained from G by reversing the edges.

Example

```
gap> G:=[ [  ], [ 4 ], [ 2 ], [ 1, 4 ] ];;
gap> ReversedGraph(G);
[ [ 4 ], [ 3 ], [  ], [ 2, 4 ] ]
```

We say that a digraph is connected when for every pair of vertices there is a path consisting of directed or reversed edges from one vertex to the other.

A.1.6 AutoConnectedComponents

▷ AutoConnectedComponents(*G*) (function)

Computes a list of the connected components of the digraph

Example

```
gap> G:=[ [  ], [ 1, 4, 5, 6 ], [  ], [ 1, 3, 5, 6 ], [ 2, 3 ], [ 2, 4, 6 ] ];;
gap> AutoConnectedComponents(G);
[ [ 1, 2, 3, 4, 5, 6 ] ]
```

Two vertices of a digraph belong to a strongly connected component if there is a directed path from each one to the other.

A.1.7 GraphStronglyConnectedComponents

▷ GraphStronglyConnectedComponents(*G*) (function)

Produces the strongly connected components of the digraph *G*.

Example

```
gap> G:=[ [  ], [ 4 ], [  ], [ 4, 6 ], [  ], [ 1, 4, 5, 6 ] ];;
gap> Set(GraphStronglyConnectedComponents(G));
[ [ 1 ], [ 2 ], [ 3 ], [ 5 ], [ 6, 4 ] ]
```

A.1.8 UnderlyingMultiGraphOfAutomaton

▷ UnderlyingMultiGraphOfAutomaton(*A*) (function)

A is an automaton. The output is the underlying directed multi-graph.

Example

```
gap> a := Automaton("det",3,"ab",[ [ 0, 3, 0 ], [ 1, 2, 3 ] ],[ 1 ],[ 1 ]);;
gap> Display(a);
   |  1  2  3
--------------
 a |     3
 b |  1  2  3
Initial state:   [ 1 ]
Accepting state: [ 1 ]
gap> UnderlyingMultiGraphOfAutomaton(a);
[ [ 1 ], [ 3, 2 ], [ 3 ] ]
```

A.1.9 UnderlyingGraphOfAutomaton

▷ UnderlyingGraphOfAutomaton(*A*) (function)

A is an automaton. The output is the underlying directed graph.

Example

```
gap> a := Automaton("det",3,"ab",[ [ 2, 3, 0 ], [ 0, 1, 3 ] ],[ 2 ],[ 1, 2, 3 ]);;
gap> Display(a);
   |  1  2  3
--------------
 a |  2  3
 b |     1  3
Initial state:    [ 2 ]
Accepting states: [ 1, 2, 3 ]
gap> UnderlyingGraphOfAutomaton(a);
[ [ 2 ], [ 1, 3 ], [ 3 ] ]
```

A.1.10 DiGraphToRelation

▷ `DiGraphToRelation(`*D*`)` (function)

Returns the relation corresponding to the digraph. Note that a directed graph may be seen in a natural way as a binary relation on the set of vertices.

Example

```
gap> G:=[ [  ], [  ], [ 4 ], [ 4 ] ];;
gap> DiGraphToRelation(G);
[ [ 3, 4 ], [ 4, 4 ] ]
```

A.1.11 MSccAutomaton

▷ `MSccAutomaton(`*A*`)` (function)

Produces an automaton where, in each strongly connected component, edges labeled by inverses are added. (M stands for modified.)

This construction is useful in Finite Semigroup Theory.

Example

```
gap> a := Automaton("det",3,"ab",[ [ 0, 2, 0 ], [ 1, 2, 0 ] ],[ 3 ],[ 1, 3 ]);;
gap> Display(a);
   |  1  2  3
--------------
 a |     2
 b |  1  2
Initial state:    [ 3 ]
Accepting states: [ 1, 3 ]
gap> MSccAutomaton(a);
< deterministic automaton on 4 letters with 3 states >
gap> Display(last);
   |  1  2  3
--------------
 a |     2
 b |  1  2
 A |     2
 B |  1  2
Initial state:    [ 3 ]
Accepting states: [ 1, 3 ]
```

A.1.12 AutoIsAcyclicGraph

▷ `AutoIsAcyclicGraph(`*G*`)` (function)

The argument is a graph's list of adjacencies and this function returns true if the argument is an acyclic graph and false otherwise.

Example

```
gap> G := [ [  ], [ 3 ], [ 2 ] ];;
gap> AutoIsAcyclicGraph(last);
false
```

Appendix B

Drawing automata

The drawing of graphs described here uses `graphviz` [DEG+02], a software for drawing graphs developed at AT & T Labs, that can be obtained at `https://www.graphviz.org/`.

B.1 Installing some external programs

In order to create the drawings you should have graphviz installed and to view them you should have installed some `pdf` viewer.

B.2 Functions to draw automata

B.2.1 DrawAutomaton

▷ `DrawAutomaton(`*`A[, state_names, L]`*`)` (function)

This function draws automaton *`A`*. The arguments *`state_names`*, *`L`* and *`file`* are optional.

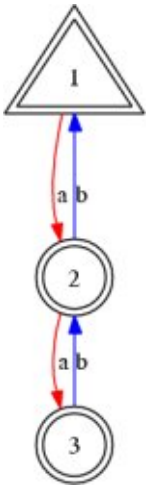
An automaton with `n` states will be drawn with numbers `1` to `n` written inside the corresponding graph node. The argument *`state_names`* is a list of `n` strings which will be the new text written inside the corresponding graph node.

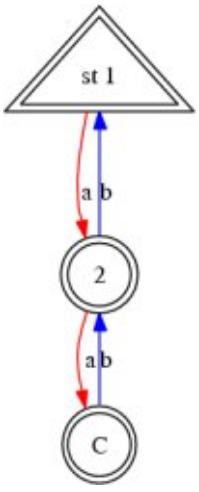
The argument *`L`* is a list of lists of integers, each of which specifies a set of states to be drawn in a different color.

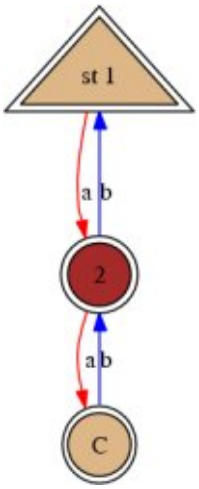
Example

```
gap> x:=Automaton("det",3,2,[ [ 2, 3, 0 ], [ 0, 1, 2 ] ],[ 1 ],[ 1, 2, 3 ]);;
gap> DrawAutomaton(x);

gap> DrawAutomaton(x,["st 1", "2", "C"]);

gap> DrawAutomaton(x,["st 1", "2", "C"],[[2],[1,3]]);
```

The output of the three previous `DrawAutomaton` commands would be the following diagrams, respectively.


1
a b
2
a b
3


st 1
a b
2
a b
C



B.2.2 DotForDrawingAutomaton

▷ `DotForDrawingAutomaton(`*arg*`)` (function)

This function computes the dot code that can be used to display an automaton. This can be done by using the function `DrawAutomaton` (B.2.1) (if the system is properly configured) or by the user in some independent way. The arguments and options are the same than those of `DrawAutomaton` (B.2.1).

Example

```
gap> DotStringForDrawingAutomaton(x);
"digraph  Automaton{\n\"1\" -> \"2\" [label=\"a\",color=red];\n\"2\" -> \"3\" \
[label=\"a\",color=red];\n\"2\" -> \"1\" [label=\"b\",color=blue];\n\"3\" -> \
\"2\" [label=\"b\",color=blue];\n\"1\" [shape=triangle,peripheries=2, style=fi\
lled, fillcolor=white];\n\"2\" [shape=doublecircle, style=filled, fillcolor=wh\
ite];\n\"3\" [shape=doublecircle, style=filled, fillcolor=white];\n}\n"
```

By using Print (or PrinTo, if one wants to print to a file) the string is displayed as follows:

Example

```
gap> Print(last);
digraph  Automaton{
"1" -> "2" [label="a",color=red];
"2" -> "3" [label="a",color=red];
"2" -> "1" [label="b",color=blue];
"3" -> "2" [label="b",color=blue];
"1" [shape=triangle,peripheries=2, style=filled, fillcolor=white];
"2" [shape=doublecircle, style=filled, fillcolor=white];
"3" [shape=doublecircle, style=filled, fillcolor=white];
}
```

The dot code produced for the remaining pictures:

Example

```
gap> Print(DotStringForDrawingAutomaton(x,["st 1", "2", "C"]));
digraph  Automaton{
"st 1" -> "2" [label="a",color=red];
"2" -> "C" [label="a",color=red];
"2" -> "st 1" [label="b",color=blue];
"C" -> "2" [label="b",color=blue];
"st 1" [shape=triangle,peripheries=2, style=filled, fillcolor=white];
"2" [shape=doublecircle, style=filled, fillcolor=white];
"C" [shape=doublecircle, style=filled, fillcolor=white];
}
gap> Print(DotStringForDrawingAutomaton(x,["st 1", "2", "C"],[[2],[1,3]]));
digraph  Automaton{
"st 1" -> "2" [label="a",color=red];
"2" -> "C" [label="a",color=red];
"2" -> "st 1" [label="b",color=blue];
"C" -> "2" [label="b",color=blue];
"st 1" [shape=triangle,peripheries=2, style=filled, fillcolor=burlywood];
"2" [shape=doublecircle, style=filled, fillcolor=brown];
"C" [shape=doublecircle, style=filled, fillcolor=burlywood];
}
```

B.2.3 DrawSubAutomaton

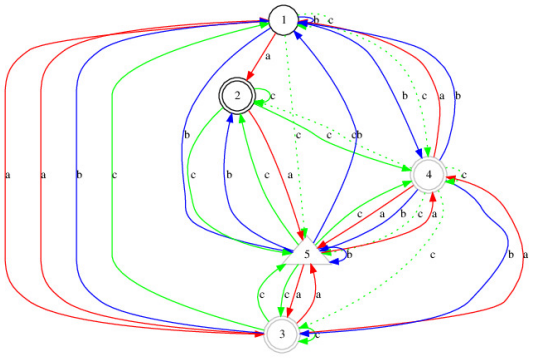
▷ `DrawSubAutomaton(`*A*`, `*B*`)` (function)

This function tests if automaton `A` is a subautomaton of `B` in which case draws `B` highlighting the edges not in `A` by drawing them in a dotted style, while the others are drawn in a plain style.

Example

```
gap> A := Automaton("nondet",5,"abc",[ [ [ 2, 3 ], [ 5 ], [ 1, 4, 5 ], [ 1, 5 ],
[ 3, 4 ] ], [ [ 1, 4, 5 ], [ ], [ 1 ], [ 1, 3, 5 ], [ 1, 2, 5 ] ], [ [ ],
[ 2, 4, 5 ], [ 1, 3, 5 ], [ ], [ 2, 3, 4 ] ] ],[ ],[ 2, 3, 4 ]);;
gap> B := Automaton("nondet",5,"abc",[ [ [ 2, 3 ], [ 5 ], [ 1, 4, 5 ], [ 1, 5 ],
[ 3, 4 ] ], [ [ 1, 4, 5 ], [ ], [ 1 ], [ 1, 3, 5 ], [ 1, 2, 5 ] ], [ [ 1, 4, 5 ],
[ 2, 4, 5 ], [ 1, 3, 5 ], [ 2, 3, 4, 5 ], [ 2, 3, 4 ] ] ],[ 3, 4, 5 ],[ 2, 3, 4 ]);;
gap> DrawSubAutomaton(A,B);
```

The output is as follows, in case the configuration of your computer permits. Otherwise, you may be interested in the dot code, as pshown below.



B.2.4 DotStringForDrawingSubAutomaton

▷ `DotStringForDrawingSubAutomaton(`*A*`, `*B*`)` (function)

This function computes the dot code that can be used to draw automata A, with the subsutomaton B emphasized. It is silently used by the function `DrawSubAutomaton` (B.2.3) (if the system is properly configured) or can be used by the user in some independent way.

Example

```
gap> DotStringForDrawingSubAutomaton(A,B);
"digraph  Automaton {\n1 -> 2 [label=\"a\",color=red];\n1 -> 3 [label=\"a\",co\
lor=red];\n1 -> 1 [label=\"b\",color=blue];\n1 -> 4 [label=\"b\",color=blue];\
\n1 -> 5 [label=\"b\",color=blue];\n1 -> 1 [label=\"c\",color=green,style = do\
tted];\n1 -> 4 [label=\"c\",color=green,style = dotted];\n1 -> 5 [label=\"c\",\
color=green,style = dotted];\n2 -> 5 [label=\"a\",color=red];\n2 -> 2 [label=\
\"c\",color=green];\n2 -> 4 [label=\"c\",color=green];\n2 -> 5 [label=\"c\",co\
lor=green];\n3 -> 1 [label=\"a\",color=red];\n3 -> 4 [label=\"a\",color=red];\
\n3 -> 5 [label=\"a\",color=red];\n3 -> 1 [label=\"b\",color=blue];\n3 -> 1 [l\
abel=\"c\",color=green];\n3 -> 3 [label=\"c\",color=green];\n3 -> 5 [label=\"c\
\",color=green];\n4 -> 1 [label=\"a\",color=red];\n4 -> 5 [label=\"a\",color=r\
ed];\n4 -> 1 [label=\"b\",color=blue];\n4 -> 3 [label=\"b\",color=blue];\n4 ->\
 5 [label=\"b\",color=blue];\n4 -> 2 [label=\"c\",color=green,style = dotted];\
\n4 -> 3 [label=\"c\",color=green,style = dotted];\n4 -> 4 [label=\"c\",color=\
green,style = dotted];\n4 -> 5 [label=\"c\",color=green,style = dotted];\n5 ->\
 3 [label=\"a\",color=red];\n5 -> 4 [label=\"a\",color=red];\n5 -> 1 [label=\"\
b\",color=blue];\n5 -> 2 [label=\"b\",color=blue];\n5 -> 5 [label=\"b\",color=\
blue];\n5 -> 2 [label=\"c\",color=green];\n5 -> 3 [label=\"c\",color=green];\n\
5 -> 4 [label=\"c\",color=green];\n3 [shape=triangle,color=gray];\n4 [shape=tr\
iangle,color=gray];\n5 [shape=triangle,color=gray];\n2 [shape=doublecircle];\n\
3 [shape=doublecircle];\n4 [shape=doublecircle];\n1 [shape=circle];\n}\n"
```

By using Print (or PrinTo, if one wants to print to a file) the string is displayed as follows:

Example

```
gap> Print(last);
digraph  Automaton {
1 -> 2 [label="a",color=red];
1 -> 3 [label="a",color=red];
```

```
1 -> 1 [label="b",color=blue];
1 -> 4 [label="b",color=blue];
1 -> 5 [label="b",color=blue];
1 -> 1 [label="c",color=green,style = dotted];
1 -> 4 [label="c",color=green,style = dotted];
1 -> 5 [label="c",color=green,style = dotted];
2 -> 5 [label="a",color=red];
2 -> 2 [label="c",color=green];
2 -> 4 [label="c",color=green];
2 -> 5 [label="c",color=green];
3 -> 1 [label="a",color=red];
3 -> 4 [label="a",color=red];
3 -> 5 [label="a",color=red];
3 -> 1 [label="b",color=blue];
3 -> 1 [label="c",color=green];
3 -> 3 [label="c",color=green];
3 -> 5 [label="c",color=green];
4 -> 1 [label="a",color=red];
4 -> 5 [label="a",color=red];
4 -> 1 [label="b",color=blue];
4 -> 3 [label="b",color=blue];
4 -> 5 [label="b",color=blue];
4 -> 2 [label="c",color=green,style = dotted];
4 -> 3 [label="c",color=green,style = dotted];
4 -> 4 [label="c",color=green,style = dotted];
4 -> 5 [label="c",color=green,style = dotted];
5 -> 3 [label="a",color=red];
5 -> 4 [label="a",color=red];
5 -> 1 [label="b",color=blue];
5 -> 2 [label="b",color=blue];
5 -> 5 [label="b",color=blue];
5 -> 2 [label="c",color=green];
5 -> 3 [label="c",color=green];
5 -> 4 [label="c",color=green];
3 [shape=triangle,color=gray];
4 [shape=triangle,color=gray];
5 [shape=triangle,color=gray];
2 [shape=doublecircle];
3 [shape=doublecircle];
4 [shape=doublecircle];
1 [shape=circle];
}
```

B.2.5 DotStringForDrawingGraph

▷ `DotStringForDrawingGraph(`*G*`)` (function)
▷ `DrawGraph(`*G*`)` (function)

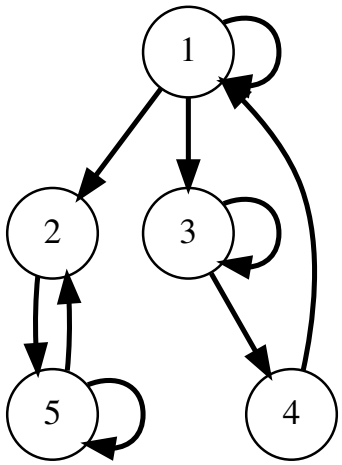
Draws a graph specified as an adjacency list `G`.

Example

```
gap> G := [[1,2,3],[5],[3,4],[1],[2,5]];
[ [ 1, 2, 3 ], [ 5 ], [ 3, 4 ], [ 1 ], [ 2, 5 ] ]
```

```
gap> Print(DotStringForDrawingGraph(G));
digraph Graph__{
1 -> 1 [style=bold, color=black];
1 -> 2 [style=bold, color=black];
1 -> 3 [style=bold, color=black];
2 -> 5 [style=bold, color=black];
3 -> 3 [style=bold, color=black];
3 -> 4 [style=bold, color=black];
4 -> 1 [style=bold, color=black];
5 -> 2 [style=bold, color=black];
5 -> 5 [style=bold, color=black];
1 [shape=circle];
2 [shape=circle];
3 [shape=circle];
4 [shape=circle];
5 [shape=circle];
}
```

The dot code can be used to produce the following picture (which may also be produced by typing `DrawGraph(G);`



B.2.6 DrawSCCAutomaton

▷ `DrawSCCAutomaton(`*`A[, state_names, L]`*`)` (function)

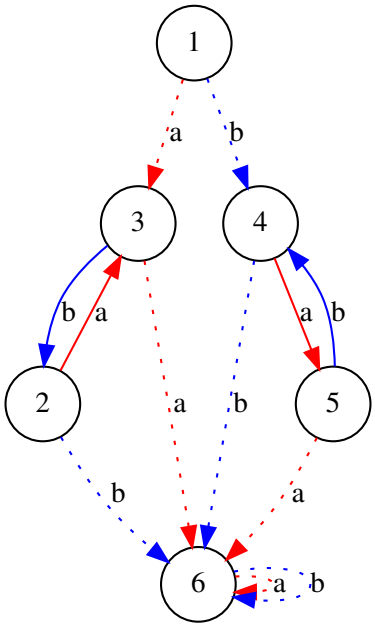
Draws automaton `A` and highlights it's strongly connected components by drawing the other edges in a dotted style.

The optional arguments *`state_names`* and *`L`* are as described in `DrawAutomaton` (B.2.1).

Example

```
gap> rcg := Automaton("det",6,"ab",[ [ 3, 3, 6, 5, 6, 6 ], [ 4, 6, 2, 6, 4, 6 ] ],
[ ],[ ]);;
gap> Print(DotStringForDrawingSCCAutomaton(rcg));
digraph  Automaton{
"1" -> "3" [label="a",color=red,style = dotted];
"2" -> "3" [label="a",color=red];
"3" -> "6" [label="a",color=red,style = dotted];
"4" -> "5" [label="a",color=red];
"5" -> "6" [label="a",color=red,style = dotted];
"6" -> "6" [label="a",color=red,style = dotted];
"1" -> "4" [label="b",color=blue,style = dotted];
"2" -> "6" [label="b",color=blue,style = dotted];
"3" -> "2" [label="b",color=blue];
"4" -> "6" [label="b",color=blue,style = dotted];
"5" -> "4" [label="b",color=blue];
"6" -> "6" [label="b",color=blue,style = dotted];
"1" [shape=circle, style=filled, fillcolor=white];
"2" [shape=circle, style=filled, fillcolor=white];
"3" [shape=circle, style=filled, fillcolor=white];
"4" [shape=circle, style=filled, fillcolor=white];
"5" [shape=circle, style=filled, fillcolor=white];
"6" [shape=circle, style=filled, fillcolor=white];
}
```

The dot code can be used to produce the following picture (which may also be produced by typing `DrawGraph(G);`



Appendix C

Inverse automata and subgroups of the free group

Inverse automata with a single initial/accepting state are in a one to one correspondence with finitely generated subgroups of the free group over the alphabet of the automaton. See [MSW01], [KM02] for details, as well as for concepts such as *flower automaton* and *Stallings foldings*.

C.1 From subgroups to inverse automata

A finitely generated subgroup of a finitely generated free group is given through a list whose first element is the number of generators of the free group and the remaining elements are the generators of the subgroup. The set of generators of the free group of rank n consists on the n first letters of the set $\{a,b,c,d,e,f,g\}$. In particular, free groups of rank greater than 8 must not be considered. A formal inverse of a generator is represented by the corresponding capital letter.

A generator of the subgroup may be given through a string of letters or through a list of positive integers as should be clear from the example that follows.

For example, `[2,"abA","bbabAB"]` stands for the subgroup of the free group of rank 2 on the generators aba^{-1} and $bbaba^{-1}b^{-1}$. The same subgroup may be given as `[2,[1,2,3],[2,2,1,2,3,4]]`. The number $n+j$ represents the formal inverse of the generator represented by j. One can go from one representation to another, using the following functions.

C.1.1 GeneratorsToListRepresentation

▷ `GeneratorsToListRepresentation(L)` (function)

Example

```
gap> L:=[2,"abA","bbabAB"];;
gap> GeneratorsToListRepresentation(L);
[ 2, [ 1, 2, 3 ], [ 2, 2, 1, 2, 3, 4 ] ]
```

C.1.2 ListToGeneratorsRepresentation

▷ `ListToGeneratorsRepresentation(K)` (function)

Example

```
gap> K:=[2,[1,2,3],[2,2,1,2,3,4]];;
gap> ListToGeneratorsRepresentation(K);
[ 2, "abA", "bbabAB" ]
```

C.1.3 FlowerAutomaton

▷ FlowerAutomaton(*L*) (function)

The argument L is a subgroup of the free group given through any of the representations described above. Returns the flower automaton.

Example

```
gap> W:=[2,"bbbAB","abAbA"];;
gap> A:=FlowerAutomaton(W);
< non deterministic automaton on 2 letters with 9 states >
gap> Display(A);
   |  1          2       3       4    5       6       7    8       9
-------------------------------------------------------------------------
 a | [ 6, 9 ]                        [ 4 ]               [ 7 ]
 b | [ 2, 5 ]   [ 3 ]   [ 4 ]               [ 7 ]       [ 9 ]
Initial state:   [ 1 ]
Accepting state: [ 1 ]
```

C.1.4 FoldFlowerAutomaton

▷ FoldFlowerAutomaton(*A*) (function)

Makes identifications on the flower automaton A. In the literature, these identifications are called Stallings foldings.

(This function may have true as a second argument. WARNING: the second argument should only be used when facilities to draw automata are available. In that case, one may visualize the identifications that are taking place.)

Example

```
gap> B := FoldFlowerAutomaton(A);
< deterministic automaton on 2 letters with 7 states >
gap> Display(B);
   |  1  2  3  4  5  6  7
-------------------------
 a |  5  4              6
 b |  2  3  4     6     5
Initial state:   [ 1 ]
Accepting state: [ 1 ]
```

C.1.5 SubgroupGenToInvAut

▷ SubgroupGenToInvAut(*L*) (function)

Returns the inverse automaton corresponding to the subgroup given by *L*.

Example

```
gap> L:=[2,"bbbAB","AbAbA"];;
gap> SubgroupGenToInvAut(L);
< deterministic automaton on 2 letters with 8 states >
gap> Display(last);
   |  1  2  3  4  5  6  7  8
-----------------------------
 a |  8  4        1     6
 b |  2  3  4     6     8
Initial state:   [ 1 ]
Accepting state: [ 1 ]
```

C.2 From inverse automata to subgroups

Given an inverse automaton with a single initial/accepting state, one can find a set of generators for the subgroup represented by the automaton. Moreover, using a geodesic tree, one can find a Nielsen reduced set of generators [KM02].

C.2.1 GeodesicTreeOfInverseAutomaton

▷ `GeodesicTreeOfInverseAutomaton(A)` (function)

Returns a subautomaton of *A* whose underlying graph is a geodesic tree of the underlying graph of the inverse automaton `A`.

Example

```
gap> A:=Automaton("det",4,2,[ [ 3, 4, 0, 0 ], [ 2, 3, 4, 0 ] ],[ 1 ],[ 1 ]);;
gap> G := GeodesicTreeOfInverseAutomaton(A);
< deterministic automaton on 2 letters with 4 states >
gap> Display(G);
   |  1  2  3  4
-----------------
 a |  3
 b |  2     4
Initial state:   [ 1 ]
Accepting state: [ 1 ]
```

C.2.2 InverseAutomatonToGenerators

▷ `InverseAutomatonToGenerators(A)` (function)

Returns a set of generators (given trough the representation above) of the subgroup of the free group corresponding to the automaton `A` given.

Example

```
gap> NW := InverseAutomatonToGenerators(A);
[ 2, "baBA", "bbA" ]
```

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analisys of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974. 6, 33

[DEG+02] D. Dobkin, J. Ellson, E. Gansner, E. Koutsofios, S. North, and G. Woodhull. Graphviz - graph drawing programs. Technical report, AT&T Research and Lucent Bell Labs, 2002. (https://www.graphviz.org). 6, 44

[HU69] J. E. Hopcroft and J. D. Ullman. *Formal Languages and their Relation to Automata*. Addison-Wesley, Reading, Mass., 1969. 33

[KM02] I. Kapovich and A. Myasnikov. Stallings foldings and subgroups of free groups. *J. of Algebra*, 248:608–668, 2002. 52, 54

[MMP+95] O. Matz, A. Miller, A. Potthoff, W. Thomas, and E. Valkema. Report on the program amore. Technical Report 9507, Christian Albrechts Universität, Kiel, 1995. 6, 32, 33

[MSW01] S. Margolis, M. Sapir, and P. Weil. Closed subgroups in pro-V topologies and the extension problem for inverse automata. *Int. J. of Algebra and Computation*, 11:405–445, 2001. 52

Index