GAP Package — KBMAG

by

Derek Holt

Mathematics Institute
University of Warwick, Coventry, CV4 7AL

Contents

1 KBMAG

KBMAG (pronounced "Kay-bee-mag") stands for **Knuth–Bendix on Monoids, and Automatic Groups**. It is a stand-alone package written in C, for use under UNIX, with an interface to GAP. This chapter describes its use as an external library from within GAP. There are interfaces for the use of KBMAG with finitely presented groups, monoids and semigroups defined within GAP. The package also contains a collection of routines for manipulating finite state automata, which can be accessed via the GAP interface.

To use this package effectively, some knowledge of the underlying theory and algorithms is advisable. The Knuth-Bendix algorithm is described in various places in the literature. Good general references that deal with the applications to groups and monoids are [LeC86] and the first few chapters of [Sim94]. For the theory of automatic groups see the multi-author book [ECH+92]. The algorithms employed by KBMAG are described more specifically in [HER91] and [Holar].

The manual for the stand-alone KBMAG package (which can be found in the `standalone/doc` directory of the package) provides more detailed information on the external C programs that are called from GAP.

Suppose that G is a finitely presented semigroup, monoid or group defined as a quotient of the free structure F. The overall objective of KBMAG is to construct a normal form for the elements of G in terms of the generators of F, together with a word reduction algorithm for calculating the normal form representative of an element in G, given by a word in the generators of F. If this can be achieved, then it is also possible to enumerate the words in normal form up to a given length, and to determine the order of G, by counting the number of words in normal form. In most serious applications, this will be infinite, since (for example) finite groups are (with some exceptions) usually handled better by Todd-Coxeter related methods. In fact a finite state automaton W is calculated that accepts precisely the language of words in the monoid generators of F that are in normal form, and W is used for the enumeration and counting functions.

The normal form of an element $g \in G$ is defined to be the least word in the generators of F (and their inverses) that represents g, with respect to a specified ordering on the set of all words in the generators of F. The available orderings are described in 1.3 below.

KBMAG offers two possible means of achieving these objectives. The first is to apply the Knuth-Bendix algorithm to the presentation, with one of the available orderings on words, and hope that the algorithm will complete with a finite confluent presentation. (If G is finite, then it is guaranteed to complete eventually but, like the Todd-Coxeter procedure, it may take a long time, or require more space than is available.) The second is to use the automatic group program, which is only applicable to groups (not to monoids or semigroups). This also uses the Knuth-Bendix procedure as one component of the algorithm, but it aims to compute certain finite state automata rather than to obtain a finite confluent rewriting system, and it completes successfully on many examples for which such a finite system does not exist. In the current standalone implementation, its use is restricted to the "shortlex" ordering on words. That is, words are ordered first by increasing length, and then words of equal length are ordered lexicographically, using the specified ordering of the generators. However, there are now some GAP procedures available in the package written by Sarah Rees that enable it be used also for the "wtlex" ordering, and the "wreathprod" ordering. See below for further details of these orderings.

For both of the above procedures, the first step is to create a GAP object known as a *Knuth-Bendix rewriting system R* from the finitely presented structure G. There are functions available that can be used to specify the input parameters for the external programs, such as the ordering on words to be used by the Knuth-Bendix procedure. One of the two external programs is then run on *R*. If successful, it updates *R*, which can then be used to reduce words in the generators of *F* to normal form, and to count and enumerate the words in normal form.

There are also now some routines available for performing corresponding operations with the cosets of a specified subgroup *H* of the group *G*. (These are not currently available for semigroups or monoids.) The words in normal form then correspond to minimal representatives under the ordering of the system of the right cosets of *H* in *G*. If successful, the index of *H* in *G* can be determined. The Knuth-Bendix routines also allow a confluent rewriting system for *H* to be computed, whereas the automatic groups routines allow a presentation of *H* to be computed (although not yet on a user-specified generating set).

In the descriptions of the functions that follow, it is important to distinguish between irreducible words, and words in normal form. As already stated, a word is in normal form if it is the least word under the ordering of the rewriting system that defines a particular group element or coset. So there is always a unique word in normal form for each group element or coset, and it is determined by the group generators and the ordering on words in the group generators. A word in a rewriting system is said to be irreducible if it does not contain the left hand side of any of the reduction rules in the system as a subword. Words in normal form are always irreducible, but the converse is true if and only if the rewriting system is confluent. The automatic groups programs provide a method of reducing words to normal form without obtaining a finite confluent rewriting system (which may not even exist).

Various levels of diagnostic output from the GAP procedures can be turned on by setting the Info variable `InfoRWS` to 1, 2 or 3.

1.1 Creating a rewriting system

First the user should be aware of a technicality. The words in a rewriting system created in GAP for use by KBMAG are defined over an alphabet that consists of the generators of a free monoid, called the `word-monoid` of the system. Suppose, as before, that the rewriting system is defined from the semigroup, monoid or group *G* which is a quotient of the free structure *F*. Then the generators of this alphabet will be in one-one correspondence with the generators (or, when *G* is a group, the generators and their inverses) of *F*, but will not be identical to them. This feature was necessary for technical reasons. Most of the user-level functions take and return words in *F* rather than the alphabet, but they do this by converting from one to the other and back.

User-level functions have also been provided to carry out this conversion explicitly if required.

The user should also be aware of a peculiarity in the way that rewriting sytems are displayed, which is really a hangover from the GAP3 interface. They are displayed nicely as a record, which gives a useful description of the system, but it does not correspond at all to the way that they are actually stored internally!

1 ► `KBMAGRewritingSystem(` *G* `)` O

constructs and returns a rewriting system *R* from a finitely presented semigroup, monoid or group *G*. When *G* is a group, the alphabet members of *R* correspond to the generators of *F* together with inverses for those generators which are not obviously involutory in *G*.

1.2 Elementary functions on rewriting systems

1 ► `IsKBMAGRewritingSystemRep(` *rws* `)` R

Returns true if *rws* is a rewriting system created by `KBMAGRewritingSystem`. The functions `IsRewritingSystem` and `IsKnuthBendixRewritingSystem` will also return true on such a system.

2 ► `IsConfluent(` *rws* `)` A

Returns true if *rws* is a rewriting system that is known to be confluent.

3 ► `SemigroupOfRewritingSytem(` *rws* `)` O

4 ► `FreeStructureOfSystem(` *rws* `)` O

5 ► `WordMonoidOfRewritingSystem(` *rws* `)` O

These return, respectively, the semigroup, monoid or group *G*, the free structure *F*, and the word-monoid of the rewriting system, as defined in the preceding section.

6 ► `ExternalWordToInternalWordOfRewritingSystem(` *rws*`,` *w* `)` F

7 ► `InternalWordToExternalWordOfRewritingSystem(` *rws*`,` *w* `)` F

These are the functions for converting between external words, which are those defined over the free structure *F* of *rws*, and the internal words, which are defined over the word-monoid of *rws*.

8 ► `Alphabet(` *rws* `)`

This is an ordered list of the generators of the word-monoid of *rws*. It will not necessarily be in the normal order of these generators, and it can be re-ordered by the function `ReorderAlphabetOfKBMAGRewritingSystem` (see below).

9 ► `Rules(` *rws* `)` O

A list of the reduction rules of *rws*. Each rule is a two-element list containing the left and right hand sides of the rule, which are words in the alphabet of *rws*.

10 ► `ResetRewritingSystem(` *rws* `)` F

This function resets the rewriting system *rws* back to its form as it was before the application of `KnuthBendix` or `AutomaticStructure`. However, the current ordering and values of control parameters will not be changed. The normal form and reduction algorithms will be unavailable after this call.

1.3 Setting the ordering

1 ► `SetOrderingOfKBMAGRewritingSystem(` *rws*`,` *ordering* `[,`*list*`] )` F

2 ► `ReorderAlphabetOfKBMAGRewritingSystem(` *rws*`,` *p* `)` F

3 ► `OrderingOfKBMAGRewritingSystem(` *rws* `)` F

4 ► `OrderingOfRewritingSystem(` *rws* `)` F

`SetOrderingOfKBMAGRewritingSystem` changes the ordering on the words of the rewriting system *rws* to *ordering*. *rws* is reset when the ordering is changed, so any previously calculated results will be destroyed. *ordering* must be one of the strings “shortlex”, “recursive”, “wtlex” and “wreathprod”. The default is “shortlex”, and this is the ordering of rewriting systems returned by `KBMAGRewritingSystem`. The orderings “wtlex” and “wreathprod” require the third parameter, *list*, which must be a list of positive integers in one-one correspondence with the alphabet of *rws* in its current order. They have the effect of attaching weights or levels to the alphabet members, in the cases “wtlex” and “wreathprod”, respectively.

Each of these orderings depends on the order of the alphabet, The current ordering of generators is displayed under the `generatorOrder` field when *rws* is viewed. This ordering can be changed by the function ’ReorderAlphabetOfKB-MAGRewritingSystem’. The second parameter *p* to this function should be a permutation that moves at most *ng* points, where *ng* is the number of generators. This permutation is applied to the current list of generators.

`OrderingOfKBMAGRewritingSystem` merely prints out a description of the current ordering.

In the “shortlex” ordering, shorter words come before longer ones, and, for words of equal length, the lexicographically smaller word comes first, using the ordering of the alphabet. The “wtlex” ordering is similar, but instead of using the length of the word as the first criterion, the total weight of the word is used; this is defined as the sum of the weights of the generators in the word. So “shortlex” is the special case of “wtlex” in which all generators have the same nonzero weight.

The “recursive” ordering is the special case of “wreathprod” in which the levels of the *ng* generators are $1,2,\ldots,ng$, in the order of the alphabet. We shall not attempt to give a complete definition of these orderings here, but refer the reader instead to pages 46–50 of [Sim94]. The “recursive” ordering is the one appropriate for a power-conjugate presentation of a polycyclic group, but where the generators are ordered in the reverse order from the usual convention for polycyclic groups. The confluent presentation will then be the same as the power-conjugate presentation. For example, for the Heisenberg group $\langle x,y,z \mid [x,z]=[y,z]=1, [y,x]=z\rangle$, a good ordering is “recursive” with the order of generators $[z^{-1},z,y^{-1},y,x^{-1},x]$. This example is included in 1.9 below.

Finally, `OrderingOfRewritingSystem` returns the appropriate GAP ordering on the elements of the word-monoid of *rws*. The standard GAP ordering functions, such as `IsLessThanUnder(`*ord*`,`*el1*`,`*el2*`)` can then be used.

1.4 Control parameters

1 ► `InfoRWS` V

This `Info` variable can be set to 0, 1, 2 or 3 to control the level of diagnostic output.

The Knuth-Bendix procedure is unusually sensitive to the settings of a number of parameters that control its operation. In some examples, a small change in one of these parameters can mean the difference between obtaining a confluent rewriting system fairly quickly on the one hand, and the procedure running on until it uses all available memory on the other hand.

Unfortunately, it is almost impossible to give even very general guidelines on these settings, although the “wreathprod” orderings appear to be more sensitive than the “shortlex” and “wtlex” orderings. The user can only acquire a feeling for the influence of these parameters by experimentation on a large number of examples.

The control parameters are defined by the user by setting values of certain fields of the *options record* of a rewriting system.

2 ► `OptionsRecordOfKBMAGRewritingSystem(` *rws* `)` F

Returns the options record *OR* of the rewriting system *rws*. The fields of *OR* listed below can be set by the user. Be careful to spell them correctly, because otherwise they will have no effect!

OR`.maxeqns`:
: A positive integer specifying the maximum number of rewriting rules allowed in *rws*. The default is 32767. If this number is exceeded, then `KnuthBendix` or `AutomaticStructure` will abort.

OR`.tidyint`:
: A positive integer, 100 by default. During the Knuth-Bendix procedure, the search for overlaps is interrupted periodically to tidy up the existing system by removing and/or simplifying rewriting rules that have become redundant. This tidying is done after finding *OR*`.tidyint` rules since the last tidying.

OR`.confnum`:
: A positive integer, 500 by default. If *OR*`.confnum` overlaps are processed in the Knuth-Bendix procedure but no new rules are found, then a fast test for confluence is carried out. This saves a lot of time if the system really is confluent, but usually wastes time if it is not.

OR`.maxstoredlen`:
: This is a list of two positive integers, *maxlhs* and *maxrhs*; the default is that both are infinite. Only those rewriting rules for which the left hand side has length at most *maxlhs* and the right hand side has length at most *maxrhs* are stored; longer rules are discarded. In some examples it is essential to impose such limits in order to obtain a confluent rewriting system. Of course, if the Knuth-Bendix procedure halts with such

limits imposed, then the resulting system need not be confluent. However, the confluence can then be tested be re-running `KnuthBendix` with the limits removed. (To remove the limits, unbind the field.)

OR`.maxoverlaplen`:
: This is apositive integer, which is infinite by default (when not set). Only those overlaps of total length *OR*`.maxoverlaplen` are processed. Similar remarks apply to those for *OR*`.maxstoredlen`.

OR`.sorteqns`:
: This should be true or false, and false is the default. When it is true, the rewriting rules are output in order of increasing length of left hand side. (The default is that they are output in the order that they were found).

OR`.maxoplen`:
: This is an integer, which is infinite by default (when not set). When it is set, the rewriting rules are output in order of increasing length of left hand side (as if *OR*`.sorteqns` were true), and only those rules having left hand sides of length up to *OR*`.maxoplen` are output at all. Again, similar remarks apply to those for *OR*`.maxstoredlen`.

OR`.maxreducelen`:
: A positive integer, 32767 by default. This is the maximum length that a word is allowed to have during the reduction process. It is only likely to be exceeded when using the “wreathprod” or “recursive” ordering.

OR`.maxstates`, *OR*`.maxwdiffs`:
: These are positive integers, controlling the maximum number of states of the word-reduction automaton used by `KnuthBendix`, and the maximum number of word-differences allowed when running `AutomaticStructure`, respectively. These numbers are normally increased automatically when required, so it unusual to want to set these flags. They can be set when either it is desired to limit these parameters (and prevent them being increased automatically), or (as occasionally happens), the number of word-differences increases too rapidly for the program to cope - when this happens, the run is usually doomed to failure anyway.

1.5 The Knuth-Bendix program

1 ▶ `KnuthBendix(` *rws* `)` O
▶ `MakeConfluent(` *rws* `)` O

These two functions do the same thing, namely to run the external Knuth-Bendix program on the rewriting system *rws*. `KnuthBendix` returns true if it finds a confluent rewriting system and otherwise false. In either case, if it halts normally, then it will update the list of the rewriting rules of *rws*, and also store a finite state automaton `ReductionAutomaton(`*rws*`)` that can be used for word reduction, and the counting and enumeration of irreducible words.

All control parameters (as defined in the preceding section) should be set before calling `KnuthBendix`. `KnuthBendix` will halt either when it finds a finite confluent system of rewriting rules, or when one of the control parameters (such as *OR*`.maxeqns`) requires it to stop. The program can also be made to halt and output manually at any time by hitting the interrupt key (normally *ctr*-’C’) once. (Hitting it twice has unpredictable consequences, since GAP may intercept the signal.)

If `KnuthBendix` halts without finding a confluent system, but still manages to output the current system and update *rws*, then it is possible to use the resulting rewriting system to reduce words, and count and enumerate the irreducible words; it cannot be guaranteed that the irreducible words are all in normal form, however. It is also possible to re-run `KnuthBendix` on the current system, usually after altering some of the control parameters. In fact, is some more difficult examples, this seems to be the only means of finding a finite confluent system.

2 ▶ `ReductionAutomaton(` *rws* `)` F

the reduction automaton of *rws*. Only expert users will wish to see this explicitly. See the section on finite state automata below for general information on functions for manipulating automata.

1.6 The automatic groups program

1 ► `AutomaticStructure(` *rws*`,` [*large*]`,` [*filestore*]`,` [*diff1*]`)` O

Run the external automatic groups program on the rewriting system *rws*. `AutomaticStructure` returns true if successful and false otherwise. If successful, it stores three finite state automata `FirstWordDifferenceAutomaton(`*rws*`)`, `SecondWordDifferenceAutomaton(`*rws*`)` and `WordAcceptor(`*rws*`)`. The first two of these are used for word-reduction, and the third for counting and enumeration of irreducible words (i.e. words in normal form).

The three optional parameters to `AutomaticStructure` are all boolean, and false by default. Setting *large* true results in some of the control parameters (such as `maxeqns` and `tidyint`) being set larger than they would be otherwise. This is necessary for examples that require a large amount of space. Setting *filestore* true results in more use being made of temporary files than would be otherwise. This makes the program run slower, but it may be necessary if you are short of core memory. Setting *diff1* to be true is a more technical option, which is explained more fully in the documentation for the stand-alone KBMAG package. It is not usually necessary or helpful, but it enables one or two examples to complete that would otherwise run out of space.

The ordering field of *rws* will usually be equal to "shortlex" for `AutomaticStructure` to be applicable. However, it is now possible to use some procedures written by Sarah Rees that work when the ordering is "wtlex" or "wreathprod". In the latter case, each generator must have the same level as its inverse.

The only control parameters for *rws* that are likely to be relevant are `maxeqns` and `maxwdiffs`.

2 ► `WordAcceptor(` *rws* `)` F

3 ► `FirstWordDifferenceAutomaton(` *rws* `)` F

4 ► `SecondWordDifferenceAutomaton(` *rws* `)` F

5 ► `GeneralMultiplier(` *rws* `)` F

These functions return, respectively, the word acceptor, the first and second word-difference automata, and the general multiplier automaton of *rws*. They can only be called after a successful call of `AutomaticStructure(`*rws*`)`. All except the word-acceptor are 2-variable automata that read pairs of words in the alphabet of *rws*. Note that the general multiplier has its states labeled, where the different labels represents the accepting states for the different letters in the alphabet of *rws*.

1.7 Word reduction

1 ► `IsReducedWord(` *rws*`,` *w* `)` A
► `IsReducedForm(` *rws*`,` *w* `)` A

These two functions do the same thing, namely to test whether the word *w* in the generators of the freestructure `FreeStructure(`*rws*`)` of the rewriting system system *rws* is reduced or not, and return true or false.

`IsReducedWord` can only be used after `KnuthBendix` or `AutomaticStructure` has been run successfully on *rws*. In the former case, if `KnuthBendix` halted without a confluent set of rules, then irreducible words are not necessarily in normal form (but reducible words are definitely not in normal form). If `KnuthBendix` completes with a confluent rewriting system or `AutomaticStructure` completes successfully, then it is guaranteed that all irreducible words are in normal form.

2 ► `ReducedForm(` *rws*`,` *w* `)` O
► `ReducedWord(` *rws*`,` *w* `)` O

Reduce the word *w* in the generators of the freestructure `FreeStructure(`*rws*`)` of the rewriting system *rws* (or, equivalently, in the generators of the underlying group of *rws*), and return the result.

`ReducedForm` can only be used after `KnuthBendix` or `AutomaticStructure` has been run successfully on *rws*. In the former case, if `KnuthBendix` halted without a confluent set of rules, then the irreducible word returned is not necessarily in normal form. If `KnuthBendix` completes with a confluent rewriting system or `AutomaticStructure` completes successfully, then it is guaranteed that all irreducible words are in normal form.

1.8 Counting and enumerating irreducible words

1 ► `Size( `*rws*` )` M

Returns the number of irreducible words in the rewriting system *rws*.

`Size` can only be used after `KnuthBendix` or `AutomaticStructure` has been run successfully on *rws*. In the former case, if `KnuthBendix` halted without a confluent set of rules, then the number of irreducible words may be greater than the number of words in normal form (which is equal to the order of the underlying group, monoid or semigroup G of *rws*). If `KnuthBendix` completes with a confluent rewriting system or `AutomaticStructure` completes successfully, then it is guaranteed that `Size` will return the correct order of G.

2 ► `Order( `*rws*`, `*w*` )` M

The order of the element w of the free structure `FreeStructure(`*rws*`)` of *rws* as an element of the group or monoid from which *rws* was defined.

`Order` can only be used after `KnuthBendix` or `AutomaticStructure` has been run successfully on *rws*. It is not guaranteed to terminate in the case of infinite order, but it usually seems to do so in practice!

3 ► `EnumerateReducedWords(`*rws*`, `*min*`, `*max*`)` O

Enumerate all irreducible words in the rewriting system *rws* that have lengths between *min* and *max* (inclusive), which should be non-negative integers. The result is returned as a list of words. The enumeration is by depth-first search of a finite state automaton, and so the words in the list returned are ordered lexicographically (not by shortlex).

`EnumerateReducedWords` can only be used after `KnuthBendix` or `AutomaticStructure` has been run successfully on *rws*. In the former case, if `KnuthBendix` halted without a confluent set of rules, then not all irreducible words in the list returned will necessarily be in normal form. If `KnuthBendix` completes with a confluent rewriting system or `AutomaticStructure` completes successfully, then it is guaranteed that all words in the list will be in normal form.

4 ► `GrowthFunction( `*rws*` )` F

Returns the growth function of the set of irreducible words in the rewriting system *rws*. This is a rational function, of which the coefficient of x^n in its Taylor expansion is equal to the number of irreducible words of length n.

If the coefficients in this rational function are larger than about 16000 then strange error messages will appear and fail will be returned.

`GrowthFunction` can only be used after `KnuthBendix` or `AutomaticStructure` has been run successfully on *rws*. In the former case, if `KnuthBendix` halted without a confluent set of rules, then not all irreducible words in the list returned will necessarily be in normal form. If `KnuthBendix` completes with a confluent rewriting system or `AutomaticStructure` completes successfully, then it is guaranteed that all words in the list will be in normal form.

1.9 Rewriting System Examples

`Example 1`

We start with a easy example - the alternating group A_4.

```
gap> F:=FreeGroup("a","b");;
gap> a:=F.1;; b:=F.2;;
gap> G:=F/[a^2, b^3, (a*b)^3];;
gap> R:=KBMAGRewritingSystem(G);
rec(
           isRWS := true,
  generatorOrder := [_g1,_g2,_g3],
        inverses := [_g1,_g3,_g2],
        ordering := "shortlex",
       equations := [
              [_g2^2,_g3],
              [_g1*_g2*_g1,_g3*_g1*_g3]
            ]
    )
    #Notice that monoid generators printed as _g1, _g2, _g3 are used
    #internally. These correspond to the group generators a, b, b^-1.
    gap> KnuthBendix(R);
    true
    gap> R;
    rec(
               isRWS := true,
         isConfluent := true,
      generatorOrder := [_g1,_g2,_g3],
            inverses := [_g1,_g3,_g2],
            ordering := "shortlex",
           equations := [
              [_g1^2,IdWord],
              [_g2*_g3,IdWord],
              [_g3*_g2,IdWord],
              [_g2^2,_g3],
              [_g3*_g1*_g3,_g1*_g2*_g1],
              [_g3^2,_g2],
              [_g2*_g1*_g2,_g1*_g3*_g1],
              [_g3*_g1*_g2*_g1,_g2*_g1*_g3],
              [_g1*_g2*_g1*_g3,_g3*_g1*_g2],
              [_g2*_g1*_g3*_g1,_g3*_g1*_g2],
              [_g1*_g3*_g1*_g2,_g2*_g1*_g3]
            ]
    )
    #The ‘equations’ field of <R> is now a complete system of rewriting rules
    gap> Size(R);
    12
    gap> EnumerateReducedWords(R,0,12);
    [ <identity ...>, a, a*b, a*b*a, a*b^-1, a*b^-1*a, b, b*a, b*a*b^-1, b^-1,
      b^-1*a, b^-1*a*b ]
    #We have enumerated all of the elements of the group - note that they
 #are returned as words in the free group F.
```

Example 2

The Fibonacci group $F(2,5)$ defined by a semigroup rather than a group presentation. Interestingly this defines the same structure (although ir would not do so for $F(2,r)$ with r even).

```
    gap> S:=FreeSemigroup(5);; a:=S.1;; b:=S.2;; c:=S.3;; d:=S.4;; e:=S.5;;
    gap> Q := S/[ [a*b,c], [b*c,d], [c*d,e], [d*e,a], [e*a,b] ];
    <fp semigroup on the generators [ s1, s2, s3, s4, s5 ]>
    gap> R:=KBMAGRewritingSystem(Q);
    rec(
               isRWS := true,
              silent := true,
      generatorOrder := [_s1,_s2,_s3,_s4,_s5],
            inverses := [,,,,],
            ordering := "shortlex",
          equations := [
            [_s1*_s2,_s3],
            [_s2*_s3,_s4],
            [_s3*_s4,_s5],
            [_s4*_s5,_s1],
            [_s5*_s1,_s2]
          ]
)
gap> KnuthBendix(R);
true
gap> Size(R);
11
gap> EnumerateReducedWords(R,0,4);
[ s1, s1^2, s1^2*s4, s1*s3, s1*s4, s2, s2^2, s2*s5, s3, s4, s5 ]
#Let's do the same thing using the "recursive" ordering.
gap> SetOrderingOfKBMAGRewritingSystem(R,"recursive");
gap> KnuthBendix(R);
true
gap> Size(R);
11
gap> EnumerateReducedWords(R,0,11);
  [ s1, s1^2, s1^3, s1^4, s1^5, s1^6, s1^7, s1^8, s1^9, s1^10, s1^11 ]
```

Example 3

The Heisenberg group - that is, the free 2-generator nilpotent group of class 2. For this to complete, we need to use the recursive ordering, and reverse our initial order of generators. (Alternatively, we could avoid this reversal, by using a "wreathprod" ordering, and setting the levels of the generators to be 6,5,4,3,2,1.)

```
gap> F:=FreeGroup("x","y","z");;
gap> x:=F.1;; y:=F.2;; z:=F.3;;
gap> G:=F/[Comm(y,x)*z^-1, Comm(z,x), Comm(z,y)];;
gap> R:=KBMAGRewritingSystem(G);
rec(
           isRWS := true,
  generatorOrder := [_g1,_g2,_g3,_g4,_g5,_g6],
        inverses := [_g2,_g1,_g4,_g3,_g6,_g5],
        ordering := "shortlex",
       equations := [
         [_g4*_g2*_g3,_g5*_g2],
         [_g6*_g2,_g2*_g6],
         [_g6*_g4,_g4*_g6]
       ]
)
gap> SetOrderingOfKBMAGRewritingSystem(R,"recursive");
gap> ReorderAlphabetOfKBMAGRewritingSystem(R,(1,6)(2,5)(3,4));
gap> R;
rec(
           isRWS := true,
  generatorOrder := [_g6,_g5,_g4,_g3,_g2,_g1],
        inverses := [_g5,_g6,_g3,_g4,_g1,_g2],
        ordering := "recursive",
       equations := [
              [_g4*_g2*_g3,_g5*_g2],
              [_g6*_g2,_g2*_g6],
              [_g6*_g4,_g4*_g6]
            ]
    )
    gap> SetInfoLevel(InfoRWS,1);
    gap> KnuthBendix(R);
    #I  Calling external Knuth-Bendix program.
    #System is confluent.
    #Halting with 18 equations.
    #I  External Knuth-Bendix program complete.
    #I  System computed is confluent.
    true
    gap> R;
    rec(
               isRWS := true,
         isConfluent := true,
      generatorOrder := [_g6,_g5,_g4,_g3,_g2,_g1],
            inverses := [_g5,_g6,_g3,_g4,_g1,_g2],
            ordering := "recursive",
           equations := [
              [_g6*_g5,IdWord],
              [_g5*_g6,IdWord],
              [_g4*_g3,IdWord],
              [_g3*_g4,IdWord],
              [_g2*_g1,IdWord],
              [_g1*_g2,IdWord],
              [_g6*_g2,_g2*_g6],
              [_g6*_g4,_g4*_g6],
              [_g4*_g2,_g2*_g4*_g5],
              [_g5*_g2,_g2*_g5],
              [_g6*_g1,_g1*_g6],
              [_g5*_g4,_g4*_g5],
              [_g6*_g3,_g3*_g6],
              [_g3*_g1,_g1*_g3*_g5],
              [_g4*_g1,_g1*_g4*_g6],
              [_g3*_g2,_g2*_g3*_g6],
              [_g5*_g1,_g1*_g5],
              [_g5*_g3,_g3*_g5]
            ]
    )
    gap> Size(R);
    infinity
    gap> IsReducedWord(R,z*y*x);
    false
    gap> ReducedForm(R,z*y*x);
    x*y*z^2
    gap> IsReducedForm(R,x*y*z^2);
 true
```

Example 4

This is an example of the use of the Knuth-Bendix algorithm to prove the nilpotence of a finitely presented group. (The method is due to Sims, and is described in Chapter 11.8 of [Sim94].) This example is of intermediate difficulty, and demonstrates the necessity of using the `maxstoredlen` control parameter.

The group is

$$\langle a,b \mid [b,a,b],[b,a,a,a],[b,a,a,a,b,a,a]\rangle$$

with left-normed commutators. The first step in the method is to check that there is a maximal nilpotent quotient of the group, for which we could use, for example, the GAP `NilpotentQuotient` command, from the package “nq”. We find that there is a maximal such quotient, and it has class 7, and the layers going down the lower central series have the abelian structures [0,0], [0], [0], [0], [0], [2], [2].

By using the stand-alone `C` nilpotent quotient program, it is possible to find a power-commutator presentation of this maximal quotient. We now construct a new presentation of the same group, by introducing the generators in this power-commutator presentation, together with their definitions as powers or commutators of earlier generators. It is this new presentation that we use as input for the Knuth-Bendix program. Again we use the recursive ordering, but this time we will be careful to introduce the generators in the correct order in the first place!

```
    gap> F:=FreeGroup("h","g","f","e","d","c","b","a");;
    gap> h:=F.1;;g:=F.2;;f:=F.3;;e:=F.4;;d:=F.5;;c:=F.6;;b:=F.7;;a:=F.8;;
    gap> G:=F/[Comm(b,a)*c^-1, Comm(c,a)*d^-1, Comm(d,a)*e^-1,
    > Comm(e,b)*f^-1, Comm(f,a)*g^-1, Comm(g,b)*h^-1,
    > Comm(g,a), Comm(c,b), Comm(e,a)];;
    gap> R:=KBMAGRewritingSystem(G);
    rec(
               isRWS := true,
      generatorOrder := [_g1,_g2,_g3,_g4,_g5,_g6,_g7,_g8,_g9,_g10,
 _g11,_g12,_g13,_g14,_g15,_g16],
            inverses := [_g2,_g1,_g4,_g3,_g6,_g5,_g8,_g7,_g10,_g9,
 _g12,_g11,_g14,_g13,_g16,_g15],
            ordering := "shortlex",
           equations := [
             [_g14*_g16*_g13,_g11*_g16],
             [_g12*_g16*_g11,_g9*_g16],
             [_g10*_g16*_g9,_g7*_g16],
             [_g8*_g14*_g7,_g5*_g14],
             [_g6*_g16*_g5,_g3*_g16],
             [_g4*_g14*_g3,_g1*_g14],
             [_g4*_g16,_g16*_g4],
             [_g12*_g14,_g14*_g12],
             [_g8*_g16,_g16*_g8]
           ]
    )
 gap> SetOrderingOfKBMAGRewritingSystem(R,"recursive");
```

A little experimentation reveals that this example works best when only those equations with left and right hand sides of lengths at most 10 are kept.

```
    gap> O:=OptionsRecordOfKBMAGRewritingSystem(R);
    gap> O.maxstoredlen:=[10,10];;
    gap> SetInfoLevel(InfoRWS,2);
    gap> KnuthBendix(R);
      # 60 eqns; total len: lhs, rhs = 129, 143; 25 states; 0 secs.
      # 68 eqns; total len: lhs, rhs = 364, 326; 28 states; 0 secs.
      # 77 eqns; total len: lhs, rhs = 918, 486; 45 states; 0 secs.
      # 91 eqns; total len: lhs, rhs = 728, 683; 58 states; 0 secs.
      # 102 eqns; total len: lhs, rhs = 1385, 1479; 89 states; 0 secs.
      . . . .
      # 310 eqns; total len: lhs, rhs = 4095, 4313; 489 states; 1 secs.
      # 200 eqns; total len: lhs, rhs = 2214, 2433; 292 states; 1 secs.
      # 194 eqns; total len: lhs, rhs = 835, 922; 204 states; 1 secs.
      # 157 eqns; total len: lhs, rhs = 702, 723; 126 states; 1 secs.
      # 151 eqns; total len: lhs, rhs = 553, 444; 107 states; 1 secs.
      # 101 eqns; total len: lhs, rhs = 204, 236; 19 states; 1 secs.
      #No new eqns for some time - testing for confluence
      #System is not confluent.
      # 172 eqns; total len: lhs, rhs = 616, 473; 156 states; 1 secs.
      # 171 eqns; total len: lhs, rhs = 606, 472; 156 states; 1 secs.
      #No new eqns for some time - testing for confluence
      #System is not confluent.
      # 151 eqns; total len: lhs, rhs = 452, 453; 92 states; 1 secs.
      # 151 eqns; total len: lhs, rhs = 452, 453; 92 states; 1 secs.
      #No new eqns for some time - testing for confluence
      #System is not confluent.
      # 101 eqns; total len: lhs, rhs = 200, 239; 15 states; 1 secs.
      # 101 eqns; total len: lhs, rhs = 200, 239; 15 states; 1 secs.
      #No new eqns for some time - testing for confluence
    #System is confluent.
    #Halting with 101 equations.
    WARNING: The monoid defined by the presentation may have changed,
             since equations have been discarded.
             If you re-run, include the original equations.
      #Exit status is 0
    #I  External Knuth-Bendix program complete.
    #WARNING: Because of the control parameters you set, the system may
    #         not be confluent. Unbind the parameters and re-run KnuthBendix
    #         to check!
    #I  System computed is NOT confluent.
    false

    #Now it is essential to re-run with the ‘maxstoredlen’ limit removed
    #to check that the system really is confluent.
    gap> Unbind(O.maxstoredlen);
    gap> KnuthBendix(R);
      # 101 eqns; total len: lhs, rhs = 200, 239; 15 states; 0 secs.
      #No new eqns for some time - testing for confluence
    #System is confluent.
    #Halting with 101 equations.
    #Exit status is 0
    #I  External Knuth-Bendix program complete.
    #I  System computed is confluent.
    true

  #In fact, in this case, we did have a confluent set already.
```

Inspection of the confluent set now reveals it to be precisely a power-commutator presentation of a nilpotent group, and so we have proved that the group we started with really is nilpotent. Of course, this means also that it is equal to its largest nilpotent quotient, of which we already know the structure.

`Example 5`

Our final example illustrates the use of the `AutomaticStructure` command, which runs the automatic groups programs. The group has a balanced symmetrical presentation with 3 generators and 3 relators, and was originally proposed by Heineken as a possible example of a finite group with such a presentation. In fact, the `AutomaticStructure` command proves it to be infinite.

This example is of intermediate difficulty, but there is no need to use any special options. It takes a few minutes to run on a WorkStation. It works better with the optional *large* parameter of `AutomaticStructure` set to `true`.

We will not attempt to explain all of the output in detail here; the interested user should consult the documentation for the stand-alone KBMAG package. Roughly speaking, it first runs the Knuth-Bendix program, which does not halt with a confluent rewriting system, but is used instead to construct a word-difference finite state automaton. This in turn is used to construct the word-acceptor and multiplier automata for the group. Sometimes the initial constructions are incorrect, and part of the procedure consists in checking for this, and making corrections. In fact, in this example, the correct automata are considerably smaller than the ones first constructed. The final stage is to run an axiom-checking program, which essentially checks that the automata satisfy the group relations. If this completes successfully, then the correctness of the automata has been proved, and they can be used for correct word-reduction and enumeration in the group.

```
gap> F:=FreeGroup("a","b","c");;
gap> a:=F.1;;b:=F.2;;c:=F.3;;
gap> G:=F/[Comm(a,Comm(a,b))*c^-1, Comm(b,Comm(b,c))*a^-1,
>                 Comm(c,Comm(c,a))*b^-1];;
gap> R:=KBMAGRewritingSystem(G);
rec(
          isRWS := true,
        verbose := true,
  generatorOrder := [_g1,_g2,_g3,_g4,_g5,_g6],
       inverses := [_g2,_g1,_g4,_g3,_g6,_g5],
       ordering := "shortlex",
      equations := [
        [_g2*_g4*_g2*_g3*_g1,_g5*_g4*_g2*_g3],
        [_g4*_g6*_g4*_g5*_g3,_g1*_g6*_g4*_g5],
        [_g6*_g2*_g6*_g1*_g5,_g3*_g2*_g6*_g1]
      ]
)
gap> SetInfoLevel(InfoRWS,1);
gap> AutomaticStructure(R,true);
#I  Calling external automatic groups program.
#Running Knuth-Bendix Program
 (pathname)/kbprog -mt 20 -hf 100 -cn 0 -wd -me 262144 -t 500 (filename)
#Halting with 42317 equations.
#First word-difference machine with 271 states computed.
#Second word-difference machine with 271 states computed.
#System is confluent, or halting factor condition holds.
```

```
#Running program to construct word-acceptor and multiplier automata
 (pathname)/gpmakefsa -l (filename)
#Word-acceptor with 1106 states computed.
#General multiplier with 2428 states computed.
#Validity test on general multiplier succeeded.
#Running program to verify axioms on the automatic structure
 (pathname)/gpaxioms -l (filename)
#General length-2 multiplier with 2820 states computed.
#Checking inverse and short relations.
#Checking relation:  _g2*_g4*_g2*_g3*_g1 = _g5*_g4*_g2*_g3
#Checking relation:  _g4*_g6*_g4*_g5*_g3 = _g1*_g6*_g4*_g5
#Checking relation:  _g6*_g2*_g6*_g1*_g5 = _g3*_g2*_g6*_g1
#Axiom checking succeeded.
#I  Computation was successful - automatic structure computed.
#Minimal reducible word acceptor with 1058 states computed.
#Minimal Knuth-Bendix equation fsa with 1891 states computed.
#Correct diff1 fsa with 271 states computed.
#Correct diff2 fsa with 271 states computed.
true

gap> Size(R);
infinity
gap> Order(R,a);
infinity
gap> Order(R,Comm(a,b));
infinity
```

1.10 Subgroups, cosets and subgroup presentations

This functions in this section are currently only applicable when the rewriting system is defined from a group *G*.

It is possible to use the Knuth-Bendix and Automatic groups program on cosets of a specified subgroup *H* of *G*. Most of the functions in the preceding sections have analogues for cosets rather than for elements. It is also possible sometimes to compute a complete rewriting system or a subgroup presentation of *H*.

1 ► `SubgroupOfKBMAGRewritingSystem(` *rws*`,` *H* `)` F

The subgroup *H* of the group *G* (= `SemigroupOfRewritingSystem(`*rws*`)`) from which *rws* is defined can be specified either as a subgroup of *G* or as a list of elements of *G* that generate *H*, or as a subgroup of *F* = `FreeStructure-OfRewritingSystem(`*rws*`)` that maps onto *H*, or as a list of elements of *F* that generate a subgroup mapping onto *H*.

`SubgroupOfKBMAGRewritingSystem` returns a rewriting system *subrws* for *H*, but *subrws* has no rules, and is only intended to be used as a parameter in the functions that follow.

2 ► `ResetRewritingSystemOnCosets(` *rws*`,` *subrws* `)` F

This function resets *subrws* back to its form as it was before the application of `KnuthBendixOnCosets` or `Automat-icStructureOnCosets`. The normal form and reduction algorithms on cosets will be unavailable after this call.

Any optional control parameters set for *rws* will automatically be used when applying the Knuth-Bendix and Automatic Structure functions on cosets, that are now to be described.

1.11 The Knuth-Bendix program on cosets

1 ► `KnuthBendixOnCosets(` *rws*, *subrws* `)` O

2 ► `KnuthBendixOnCosetsWithSubgroupRewritingSystem(` *rws*, *subrws* `)` O

Run the external Knuth-Bendix program on the rewriting system *rws* with respect to the cosets of the subgroup corresponding to *subrws*. `KnuthBendixOnCosets` returns true if it finds a confluent rewriting system on coset representatives, and otherwise false.

If `KnuthBendixOnCosets` halts without finding a confluent system, but still manages to output the current system and update *rws*, then it is possible to use the resulting rewriting system to reduce coset representatives, and count and enumerate the irreducible coset representatives; it cannot be guaranteed that the irreducible coset representatives are all in normal form, however.

`KnuthBendixOnCosetsWithSubgroupRewritingSystem` does the same and, in addition, tries to compute a confluent rewriting system for the subgroup *H*.

3 ► `RewritingSystemOfSubgroupOfKBMAGRewritingSystem(` *rws*, *subrws* `)` F

This can only be used after a successful call of `KnuthBendixOnCosetsWithSubgroupRewritingSystem`. It returns a confluent rewriting system for *H* on a generating set corresponding to the generators of *H* that were used to define *subrws*.

1.12 The automatic cosets program

1 ► `AutomaticStructureOnCosets(`*rws*, *subrws*, `[`*large*`]`, `[`*filestore*`]`, `[`*diff1*`])` O

2 ► `AutomaticStructureOnCosetsWithSubgroupPresentation(`*rws*, *subrws*, `[`*large*`]`, `[`*filestore*`]`, `[`*diff1*`])` O

Run the external automatic cosets program on the rewriting system *rws* with respect to the cosets of the subgroup *H* from which *subrws* was defined. `AutomaticStructureOnCosets` returns true if successful and false otherwise.

The optional parameters to `AutomaticStructureOnCosets` are the same as for `AutomaticStructure`.

The ordering of *rws* must be “shortlex”.

`AutomaticStructureOnCosetsWithSubgroupPresentation` does the same and, in addition, tries to compute a presentation of the subgroup *H*.

3 ► `PresentationOfSubgroupOfKBMAGRewritingSystem(` *rws*, *subrws* `)` F

This can only be used after a successful call of `AutomaticStructureOnCosetsWithSubgroupPresentation`. It returns a presentation for *H*, but this is not on the generators used to define *H*.

1.13 Word reduction on cosets

1 ► `IsReducedCosetRepresentative(` *rws*, *subrws*, *w* `)` A

Test whether the word *w* in the generators of the freestructure `FreeStructure(`*rws*`)` of the rewriting system system *rws* is reduced or not as a coset representative of the subgroup *H* of *G*, and return true or false.

`IsReducedCosetRepresentative` can only be used after `KnuthBendixOnCosets` or `AutomaticStructureOn-Cosets` has been run successfully on *rws* and *subrws*. In the former case, if `KnuthBendixOnCosets` halted without a confluent set of rules, then irreducible words are not necessarily in normal form (but reducible words are definitely not in normal form). If `KnuthBendixOnCosets` completes with a confluent rewriting system or `AutomaticStruc-tureOnCosets` completes successfully, then it is guaranteed that all irreducible words are in normal form.

2 ► `ReducedFormOfCosetRepresentative(` *rws*`,` *subrws*`,` *w* `)` O
► `ReducedCosetRepresentative(` *rws*`,` *subrws*`,` *w* `)` O

Reduce the word *w* in the generators of the free structure `FreeStructure(`*rws*`)` of the rewriting system *rws* as a coset representative of the subgroup *H* from which *subrws* was defined, and return the result.

`ReducedFormOfCosetRepresentative` can only be used after `KnuthBendixOnCosets` or `AutomaticStructure-OnCosets` has been run successfully on *rws* and *subrws*. In the former case, if `KnuthBendixOnCosets` halted without a confluent set of rules, then the irreducible word returned is not necessarily in normal form. If `KnuthBendixOn-Cosets` completes with a confluent rewriting system or `AutomaticStructureOnCosets` completes successfully, then it is guaranteed that all irreducible words are in normal form.

1.14 Counting and enumerating irreducible words for cosets

1 ► `Index(` *rws*`,` *subrws* `)` M

Returns the number of irreducible words for coset represenatitives of the subgroup *H* of *G* corresponding to *subrws*.

`Index` can only be used after `KnuthBendixOnCosets` or `AutomaticStructureOnCosets` has been run successfully on *rws* and *subrws*. In the former case, if `KnuthBendixOnCosets` halted without a confluent set of rules, then the number of irreducible words may be greater than the number of words in normal form (which is equal to the index of *H* in *G*). If `KnuthBendixOnCosets` completes with a confluent rewriting system or `AutomaticStructureOnCosets` completes successfully, then it is guaranteed that `Index` will return the correct index of *H* in *G*.

2 ► `EnumerateReducedCosetRepresentatives(`*rws*`,` *subrws*`,` *min*`,` *max*`)` O

Enumerate all irreducible words for coset representatives of *H* in *G*, that have lengths between *min* and *max* (inclusive), which should be non-negative integers. The result is returned as a list of words. The enumeration is by depth-first search of a finite state automaton, and so the words in the list returned are ordered lexicographically (not by shortlex).

`EnumerateReducedCosetRepresentatives` can only be used after `KnuthBendixOnCosets` or `AutomaticStruc-tureOnCosets` has been run successfully on *rws* and *subrws*. In the former case, if `KnuthBendixOnCosets` halted without a confluent set of rules, then not all irreducible words in the list returned will necessarily be in normal form. If `KnuthBendixOnCosets` completes with a confluent rewriting system or `AutomaticStructureOnCosets` completes successfully, then it is guaranteed that all words in the list will be in normal form.

3 ► `GrowthFunctionOfCosetRepresentatives(` *rws*`,` *subrws* `)` F

Returns the growth function of the set of irreducible words for coset representatives of *H* in *G*, where *subrws* and *rws* are the rewriting systems for *H* and *G*. This is a rational function, of which the coefficient of x^n in its Taylor expansion is equal to the number of coset representatives words of length *n*.

If the coefficients in this rational function are larger than about 16000 then strange error messages will appear and fail will be returned.

`GrowthFunctionOfCosetRepresentatives` can only be used after `KnuthBendixOnCosets` or `AutomaticStruc-tureOnCosets` has been run successfully on *rws* and *subrws*. In the former case, if `KnuthBendixOnCosets` halted without a confluent set of rules, then not all irreducible words in the list returned will necessarily be in normal form. If `KnuthBendixOnCosets` completes with a confluent rewriting system or `AutomaticStructureOnCosets` completes successfully, then it is guaranteed that all words in the list will be in normal form.

1.15 Examples of the Use of Rewriting System On Cosets

`Example 1`

```
  gap> F:=FreeGroup("a","b","c");;
  gap> a:=F.1;;b:=F.2;;c:=F.3;;
  gap> G := F/[b^3,c^3,(b*c)^4,(b*c^-1)^5,a^-1*b^-1*c*b*c*b^-1*c*b*c^-1];
  <fp group on the generators [ a, b, c ]>
  gap> R:=KBMAGRewritingSystem(G);
  rec(
             isRWS := true,
            silent := true,
    generatorOrder := [_g1,_g2,_g3,_g4,_g5,_g6],
          inverses := [_g2,_g1,_g4,_g3,_g6,_g5],
          ordering := "shortlex",
         equations := [
           [_g3^2,_g4],
           [_g5^2,_g6],
           [_g3*_g5*_g3*_g5,_g6*_g4*_g6*_g4],
           [_g3*_g6*_g3*_g6*_g3,_g5*_g4*_g5*_g4*_g5],
           [_g2*_g4*_g5*_g3*_g5,_g5*_g4*_g6*_g3]
         ]
  )
  gap> S:=SubgroupOfKBMAGRewritingSystem(R,[a^3,c*a^2]);
  rec(
             isRWS := true,
            silent := true,
    generatorOrder := [_x1,_X1,_x2,_X2],
          inverses := [_X1,_x1,_X2,_x2],
          ordering := "shortlex",
         equations := [
         ]
  )
  gap> KnuthBendixOnCosetsWithSubgroupRewritingSystem(R,S);
  true
  gap> Index(R,S);
  18
  gap> IsReducedCosetRepresentative(R,S,b*a*b*a);
  false
  gap> ReducedFormOfCosetRepresentative(R,S,b*a*b*a);
  b^-1*a^-1
  gap> EnumerateReducedCosetRepresentatives(R,S,0,4);
  [ <identity ...>, a, a*b, a*b*c, a*b^-1, a^-1, a^-1*b, a^-1*b*c, a^-1*b^-1,
    b, b*c, b*c*a, b*c*a^-1, b*c^-1, b^-1, b^-1*a, b^-1*a^-1, b^-1*a^-1*b ]
  gap> SS:=RewritingSystemOfSubgroupOfKBMAGRewritingSystem(R,S);;
  gap> Size(SS);
60
```

`Example 2`

We find a free subgroup of the Fibonacci group $F(2,8)$. This example may take about 20 minutes to run on a typical WorkStation.

```
gap> F:=FreeGroup(8);;
gap> a:=F.1; b:=F.2; c:=F.3; d:=F.4; e:=F.5; f:=F.6; g:=F.7; h:=F.8;
gap> G := F/[a*b*c^-1, b*c*d^-1, c*d*e^-1, d*e*f^-1,
>           e*f*g^-1, f*g*h^-1, g*h*a^-1, h*a*b^-1];
gap> R:=KBMAGRewritingSystem(G);;
gap> S:=SubgroupOfKBMAGRewritingSystem(R,[a,e]);;
gap> AutomaticStructureOnCosetsWithSubgroupPresentation(R,S);
gap> P:=PresentationOfSubgroupOfKBMAGRewritingSystem(R,S);
<fp group on the generators [ f1, f3 ]>
gap> RelatorsOfFpGroup(P);
[  ]
gap> Index(R,S);
infinity
```

1.16 Functions for manipulating finite state automata

The KBMAG package contains GAP interfaces to many of the functions for manipulating finite state automata (fsa) that are available in the standalone. We shall list these here, without giving much detail. For more detail, the user could try looking in the source file `gap/fsa4.g`. fsa are currently implemented as GAP records, as they were previously in GAP3. This interface may be updated to the style of GAP4 at some stage. (Note that the abbreviation fsa will be used for both singular and the plural.)

The alphabet of an fsa is itself a record that must contain at least the two components `type` and `size`, where `type` is a string, and `size` a positive integer. The easiest possibility is to use the type “simple”, and then no other record components are necessary. There are several more complicated possibilities, which are used by the other rewriting system functions. For example, there is the type “identifiers”, for which fields “format” and “names” are necessary. For example

```
gap> M:=FreeMonoid(3);;
gap> alph := rec(type:="identifiers", size:=3,
           format:="dense", names:=[M.1,M.2,M.3] );;
```

defines a valid alphabet for an fsa. The members of the alphabet are referred to as `letters`, and can be represented either by a positive integer or by their name (usually a generator of a free group or monoid) if they have one.

The functions `ReductionAutomaton(`*rws*`)`, `WordAcceptor(`*rws*`)`, `FirstWordDifferenceAutomaton(`*rws*`)`, `SecondWordDifferenceAutomaton(`*rws*`)` and `GeneralMultiplie(`*rws*`)` mentioned in earlier sections all return a fsa. The other possibilities for the user to construct a fsa are to use the function `FSA` or to read one in from a file. In the latter case, the user must immediately call `InitializeFSA` on the record that has been read in. For example, running GAP from the package directory:

```
    gap> Read("standalone/fsa_data/fsa_2");
  gap> InitializeFSA(fsa_2);
```

1 ► `IsInitializedFSA(` *fsa* `)` F

Tests whether *fsa* is a record describing a valid initialized fsa.

2 ► `InitializeFSA(` *fsa* `)` F

Initializes a record representing a fsa that has been read in from a file.

3 ► `FSA(` *alph* `)` F

Returns an initialized fsa with alphabet *alph* having one state that is an initial and final state, and no transitions (edges).

The arguments of the following functions must be initialized fsa.

4 ▶ `WriteFSA(` *fsa* `)`

Displays *fsa* nicely. (In a proper implementation, this would be the `ViewObj` function.)

5 ▶ `IsDeterministicFSA(` *fsa* `)` F

Tests whether *fsa* is a deterministic fsa. Many of the functions below work only for deterministic fsa. A deterministic fsa with the same language as *fsa* can be constructed with the function `DeterminizeFSA`.

6 ▶ `AlphabetFSA(` *fsa* `)` F

7 ▶ `StatesFSA(` *fsa* `)` F

Return, respectively, the records representing the alphabet and state-set of *fsa*.

8 ▶ `NumberOfStatesFSA(` *fsa* `)` F

Returns the number of states of *fsa*.

9 ▶ `NumberOfLettersFSA(` *fsa* `)` F

10 ▶ `SizeOfAlphabetFSA(` *fsa* `)` F

Returns the size of the alphabet of *fsa*.

11 ▶ `AcceptingStatesFSA(` *fsa* `)` F

Returns the list of accepting states of *fsa*.

12 ▶ `InitialStatesFSA(` *fsa* `)` F

Returns the list of initial states of *fsa*.

13 ▶ `DenseDTableFSA(` *fsa* `)` F

fsa must be deterministic. The transition table is returned as a list of lists, where the e-th entry in the s-th inner list is `TargetDFA(`*fsa*`,`*e*`,`*s*`)` (see below).

14 ▶ `SparseTableFSA(` *fsa* `)` F

The transition table is returned as a list of lists, where each entry in the s-th inner list is is a two-element list `[`*e*`,`*t*`]` of integers that represents a transition from state number s to state number t under letter number e. We can also have e=0, representing transitions with no label (ε transitions).

15 ▶ `TargetDFA(` *fsa*`,` *e*`,` *s* `)` F

fsa must be a deterministic fsa, e should be a number or name of a letter of the alphabet, and s a number of a state of *fsa*. The target of s under e is returned, or 0 if there is no target.

16 ▶ `TargetsFSA(` *fsa*`,` *e*`,` *s* `)` F

Similar to `TargetDFA`, but *fsa* need not be deterministic, and a list of targets is returned.

17 ▶ `SourcesFSA(` *fsa*`,` *e*`,` *s* `)` F

Similar to `TargetsFSA`, but the list of sources of transitions to s under e is returned. e can also be zero here.

18 ▶ `WordTargetDFA(` *fsa*`,` *w* `)` F

fsa must be a deterministic fsa, and w should be a list of integers or a word in the alphabet (in the case when the alphabet is defined from a free group or monoid). The target of the initial state of *fsa* under w is returned, or 0 if there is no such target.

19 ► `IsAcceptedWordDFA(` *fsa*, *w* `)` F

fsa must be a deterministic fsa, and *w* should be a list of integers or a word in the alphabet (in the case when the alphabet is defined from a free group or monoid). This function tests whether *w* is in the language of *fsa*.

20 ► `AddStateFSA(` *fsa* `)` F

Adds an extra non-accepting state to *fsa* with no transitions to or from it.

21 ► `DeleteStateFSA(` *fsa* `)` F

Deletes the highest numbered state together with all transitions to and from it from *fsa*. Use `PermuteStatesFSA` first to delete a different state.

22 ► `PermuteStatesFSA(` *fsa*, *p* `)` F

p should be a permutation of `[1..ns]`, where *fsa* has *ns* states. The states are permuted, where the original state number n becomes the new state number n^p.

23 ► `AddLetterFSA(` *fsa* `[,`*name*`]` `)` F

Adds an extra letter to the alphabet of *fsa* with no associated transitions. If the alphabet of *fsa* is defined over a free group or monoid, then *name* specifies the element of this free structure corresponding to the new letter.

24 ► `DeleteLetterFSA(` *fsa* `)` F

Deletes the highest numbered letter together with all associated transitions from the alphabet of *fsa*. Use `PermuteLettersFSA` first to delete a different letter.

25 ► `PermuteLettersFSA(` *fsa*, *p* `)` F

p should be a permutation of `[1..na]`, where the alphabet of *fsa* has *na* letters. The letters are permuted, where the original letter number n becomes the new letter number n^p.

26 ► `AddEdgeFSA(` *fsa*, *e*, *s*, *t* `)` F

Adds an extra transition to *fsa* with source *s*, target *t* and letter *e*. If there is already such a transition, then this function does nothing.

27 ► `DeleteEdgeFSA(` *fsa*, *e*, *s*, *t* `)` F

Deletes the transition from *fsa* with source *s*, target *t* and letter *e* if there is one.

28 ► `SetAcceptingFSA(` *fsa*, *s*, *flag* `)` F

flag should be true or false, and state number *s* is made accepting or non-accepting, respectively.

29 ► `SetInitialFSA(` *fsa*, *s*, *flag* `)` F

flag should be true or false, and state number *s* is made initial or non-initial, respectively.

30 ► `IsAccessibleFSA(` *fsa* `)` F

Tests whether *fsa* is accessible; that is, whether all states can be reached from the initial states.

31 ► `AccessibleFSA(` *fsa* `)` F

Removes all inaccessible states from *fsa* thus rendering it accessible without altering its language.

32 ► `IsTrimFSA(` *fsa* `)` F

Tests whether *fsa* is trim; that is, whether all states can be reached from the initial states, and a final state can be reached from all states.

33 ▶ `TrimFSA(` *fsa* `)` F

Removes all inaccessible states from *fsa* and all states from which an accepting state cannot be reached, thus rendering it trim without altering its language.

34 ▶ `IsBFSFSA(` *fsa* `)` F

Tests whether *fsa* is in `bfs` (breadth-first-search) form; that is, whether the initial states come first and the other states appear in ascending order if one scans the transition table first by state number and then by alphabet number. Note that *fsa* must be accessible for it to be `bfs`.

35 ▶ `BFSFSA(` *fsa* `)` F

Replaces *fsa* by one with the same language but in `bfs` form. This can be useful for comparing the languages of two fsa. In fact `MinimizeFSA` and `BFSFSA` are applied in turn to a deteministic *fsa*, then the resulting transition table is uniquely determined by the language of the *fsa*.

36 ▶ `LSizeDFA(` *fsa* `)` F

The size of the acceted language of *fsa*, which must be deterministic. This only works if *fsa* is trim. If not, then `TrimFSA` must be called first.

37 ▶ `LEnumerateDFA(` *fsa*`,` *min*`,` *max* `)` F

The words in the language of *fsa* of length l satisfying $min \leq l \leq max$ are returned in a list. The words will either be lists of integers, for “simple” type alphabets, of lists of words in the underlying free group or monoid for alphabets of type “identifiers”.

The remaining fsa functions all call external programs from the standalone. All of them except `DeterminizeFSA` take only deterministic fsa as input, and all of them return deterministic fsa as output.

38 ▶ `DeterminizeFSA(` *fsa* `)` F

Returns a deterministic fsa with the same language as *fsa*.

39 ▶ `MinimizeFSA(` *fsa* `)` F

Returns a fsa with the same language as *fsa* and a minimal number of states.

40 ▶ `NotFSA(` *fsa* `)` F

Returns a fsa with the same alphabet as *fsa* whose language is the complement of that of *fsa*.

41 ▶ `StarFSA(` *fsa* `)` F

Returns a fsa whose language is L^*, where L is the langauge of *fsa*.

42 ▶ `ReverseFSA(` *fsa* `)` F

Returns a fsa whose language consists of the reversed words in the language of *fsa*.

43 ▶ `ExistsFSA(` *fsa* `)` F

fsa should be two-variable fsa on an alphabet A. An fsa is returned that accepts a word $w_1 \in A^*$ if and only if there exists a words $w_2 \in A^*$ with (w_1, w_2) in the language of *fsa*.

44 ▶ `SwapCoordsFSA(` *fsa* `)` F

fsa should be two-variable fsa on an alphabet A. A two-variable fsa on A is returned that accepts (w_1, w_2) if and only if (w_2, w_1) is accepted by *fsa*.

45 ▶ `AndFSA(` *fsa1*`,` *fsa2* `)` F

fsa1 and *fsa2* must have the same alphabet. The returned fsa has language equal to the interssection of those of *fsa1* and *fsa2*.

46 ► `OrFSA(` *fsa1*`,` *fsa2* `)` F

fsa1 and *fsa2* must have the same alphabet. The returned fsa has language equal to the union of those of *fsa1* and *fsa2*.

47 ► `ConcatFSA(` *fsa1*`,` *fsa2* `)` F

fsa1 and *fsa2* must have the same alphabet. The returned fsa accepts words of the form w_1w_2, where w_1 and w_2 are in the languages of *fsa1* and *fsa2*, respectively.

48 ► `LanguagesEqualFSA(` *fsa1*`,` *fsa2* `)` F

fsa1 and *fsa2* must have the same alphabet. This function tests whether the languages of *fsa1* and *fsa2* are equal, and returns True or false.

49 ► `GrowthFSA(` *fsa* `)` F

Returns the growth function of *fsa*. This is a rational function, of which the the coefficient of x^n in its Taylor expansion is equal to the number of words of length n in the accepted language of *fsa*.

If the coefficients in this rational function are larger than about 16000 then strange error messages will appear and fail will be returned.

Bibliography

[ECH+92] D.B.A. Epstein, J.W. Cannon, D.F. Holt, S. Levy, M.S. Paterson, and W.P. Thurston. *Word Processing and Group Theory*. Jones and Bartlett, 1992.

[HER91] D.F. Holt, D.B.A. Epstein, and S. Rees. The use of knuth-bendix methods to solve the word problem in automatic groups. *J. Symbolic Computation*, 12:397–414, 1991.

[Holar] Derek F. Holt. The warwick automatic groups software. In *Proceedings of DIMACS Conference on Computational Group Theory, Rutgers, March 1994.*, To appear.

[LeC86] P. LeChenadec. *Canonical Forms in Finitely Presented Algebras*. London Pitman and New York, Wiley, 1986.

[Sim94] Charles C. Sims. *Computation with Finitely Presented Groups*. Cambridge, 1994.