AutomGrp

—

A GAP4 Package

Version 1.3

by

Yevgen Muntyan

Bellevue, WA, USA

Dmytro Savchuk

Department of Mathematics and Statistics

University of South Florida, Tampa, FL, 33620, USA

muntyan@fastmail.fm

dmytro.savchuk@gmail.com

http://finautom.sourceforge.net/

March 2016

Contents

1 Introduction

The `AutomGrp` package provides methods for computation with groups and semigroups generated by finite automata or given by wreath recursions, as well as with their finitely generated subgroups, subsemigroups and elements.

The project originally started in 2000 mostly for personal use. It was gradually expanding during consequent years, including both addition of new algorithms and simplification of user interface. It was used in the process of classification of groups generated by 3-state automata over a 2-letter alphabet (see [BGK+08]), as well as many subsequent papers.

First author thanks Sveta and Max Muntyan for their infinite patience and understanding. Second author thanks Olga, Anna, Irina and Andrey Savchuk for their help and understanding. This project would be impossible without them.

We would like to express our warm gratitude to Rostislav Grigorchuk, Zoran Sunic, Volodymyr Nekrashevych, Ievgen Bondarenko, Rostyslav Kravchenko, Yaroslav and Maria Vorobets and Ben Steinberg for their support, valuable comments, feature requests and constant interest in the project.

We also appreciate the code provided by Andrey Russev that was used to optimize the minimization of autmata function. Last, but not the least, we want to thank the anonymous referee for a very detailed review and numeruous constructive comments that significantly increased the quality of the accepted version of the package.

Both authors were partially supported by NSF grants DMS-0600975, DMS-0456185 and DMS-0308985. The second author appreciates the support from the New Researcher Grant from University of South Florida and the Simons Collaboration Grant from Simons Foundation.

1.1 Short math background

This package deals mostly with groups acting on rooted trees. In this section we recall necessary definitions and notation that will be used throughout the manual. For more detailed introduction to the theory of groups generated by automata we refer the reader to [GNS00].

The infinite connected tree with selected vertex, called the *root*, in which the degree of every vertex except the root is $d+1$ and the degree of the root is d is called the *regular homogeneous rooted tree of degree d* (or *d-ary tree*). The rooted tree of degree 2 is called the *binary tree*.

The n-th *level* of the tree consists of all vertices located at distance n from the root (here we mean combinatorial distance in the graph).

Similarly one defines *spherically homogeneous* (or *spherically-transitive*) rooted trees as rooted trees, such that the degrees of all vertices at each level coincide (but may depend on the level).

Given a finite alphabet $X = \{1, 2, \dots, d\}$ the set X^* of all finite words over X may be endowed with the structure of a d-ary tree in which the empty word $\emptyset$ is the *root*, the *level* n in X^* consists of the words of length n over X and every vertex v has d children, labeled by vx, for $x \in X$.

Any automorphism f of a rooted tree T fixes the root and the levels. For any vertex v of the tree T each automorphism f induces the automorphism $f|_v$ of the subtree of T hanging down from the vertex v by

$f|_v(u) = w$ if $f(vu) = v'w$ for some $v' \in X^{|v|}$ from the same level as v (here $|v|$ denotes the combinatorial distance from v to the root of the tree). This automorphism is called the *section* of f at v.

If the tree T is regular, then the subtrees hanging down from vertices of T are canonically isomorphic to T and, thus, the sections of any automorphism f of T can be considered as automorphisms of T again.

A group G of automorphisms of a regular rooted tree T is called *self-similar* if all sections of every element of G belong to G.

A self-similar group G is called *contracting* if there is a finite set N of elements of G, such that for any g in G there is a level n such that all sections of g at vertices of levels bigger than n belong to N. The smallest set with such a property is called the *nucleus* of G.

Any automorphism f of a rooted tree can be decomposed as

$$f = (f_1, f_2, \ldots, f_d)\sigma,$$

where $f_1, \ldots, f_d$ are the sections of f at the vertices of the first level and σ is the permutation which permutes the subtrees hanging down from these vertices.

This notation is very convenient for performing multiplication of elements. Throughout this manual and everywhere in the package we use the right action of (semi)groups acting on trees and for automorphisms (homomorphisms) of a tree f and g, the composition $f \cdot g$ means that f acts first. This is consistent with the order of multiplication of permutations in GAP , even though some authors in the field prefer to use the left action. With this convention in mind, If $f = (f_1, f_2, \ldots, f_d)\sigma$ and $g = (g_1, g_2, \ldots, g_d)\pi$, then

$$f \cdot g = (f_1 \cdot g_{\sigma(1)}, \ldots, f_d \cdot g_{\sigma(d)})\sigma\pi,$$

$$f^{-1} = (f^{-1}_{\sigma^{-1}(1)}, \ldots, f^{-1}_{\sigma^{-1}(d)})\sigma^{-1}.$$

The group of automorphisms of a rooted tree is said to be *level-transitive* if it acts transitively on each level of the tree.

Everything above applies also for homomorphisms of rooted trees (maps preserving the root and incidence relation of the vertices). The only difference is that in this case we get semigroups and monoids of tree homomorphisms.

A special class of self-similar groups is the class of groups generated by finite automata. This class is especially nice from the algorithmic point of view. Let us recall basic definitions.

A *Mealy automaton* (*transducer*, *synchronous automaton*, or, simply, *automaton*) is a tuple $A = (Q, X, \rho, \tau)$, where Q is a set of *states*, X is a finite *alphabet* of cardinality $d \geq 2$, $\rho : Q \times X \to X$ is a map, called the *output map*, $\tau : Q \times X \to Q$ is a map, called the *transition map*.

If for each state q in Q, the restriction $\rho_q : X \to X$ given by $\rho_q(x) = \rho(q, x)$ is a permutation, the automaton is called *invertible*.

If the set Q of states is finite, the automaton is called *finite*.

If some state q in Q of the automaton A is selected to be initial, the automaton is called *initial* and denoted A_q. If an initial state is not specified, the automaton is called *noninitial*.

An initial automaton naturally acts on X^* by homomorphisms (automorphisms in the case of an invertible automation) in the following way. Given a word $x_1x_2 \ldots x_n$, the automaton starts at the initial state q, reads the first input letter x_1, outputs the letter $\rho_q(x_1)$ and changes its state to $q_1 = \tau(q, x_1)$. The rest of the input word is handled by the new state q_1 in the same way. Formally speaking, the functions ρ and τ can be extended to ρ: $Q \times X^* \to X^*$ and τ: $Q \times X^* \to Q$.

Given an automaton A the group $G(A)$ of automorphisms of X^* generated by all the states of A (as initial automata) is called the *automaton group* defined by A.

Every automaton group is self-similar, because the section of A_q at vertex v is just $A_{\tau(q,v)}$.

A special case is the case of groups generated by finite automata and their subgroups. In this class we can solve the word problem, which makes it much nicer from the computational point of view.

Finite automata are often described by *recursive relations* (or *wreath recursion*) of the form

$$q = (\tau(q, 1), \ldots, \tau(q, d))\rho_q$$

for every state q. For example, the line $a = (a, b)(1, 2), b = (a, b)$ describes the automaton with 2 states a and b, such that a permutes the letters 1 and 2 of the alphabet $X = \{1, 2\}$ and b does not; and, independently of the current state, the automaton changes its initial state to a if it reads 1 and to b if it reads 2. This particular automaton generates the, so-called, lamplighter group.

Semigroups generated by noninvertible automata are defined in a similar way.

1.2 Installation instructions

`AutomGrp` package requires GAP version at least 4.4.6 and `FGA` (Free Group Algorithms) package available at

```
http://www.gap-system.org/Packages/fga.html
```

The installation of the `AutomGrp` package follows the standard GAP rules, i.e. to install it unpack the archive into the `pkg` directory of your GAP distribution. This will create `automgrp` subdirectory.

To load package issue the command

```
gap> LoadPackage("automgrp");
-----------------------------------------------------------------
Loading  AutomGrp 1.3 (Automata Groups and Semigroups)
by Yevgen Muntyan (muntyan@fastmail.fm)
   Dmytro Savchuk (http://savchuk.myweb.usf.edu/)
Homepage: http://finautom.sourceforge.net/
-----------------------------------------------------------------
true
```

Note, that if the `FR` package by Laurent Bartholdi is installed as well, GAP will automatically load it, together with the packages on which it depends. The `FR` package functionality partially overlaps with the one of `AutomGrp`. For the user's convenience and to expand the functionality of both packages, several converters (see operations `AutomGrp2FR` (5.1.2) and `FR2AutomGrp` (5.1.1)) of the basic data types were implemented in `AutomGrp`.

To test the installation, issue the command

```
gap> Read( Filename( DirectoriesLibrary( "pkg/automgrp/tst"), "testall.g"));
```

in the GAP command line.

1.3 Quick example

Here is how to define the Grigorchuk group and Basilica group.

```
gap> Grigorchuk_Group := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
```

Similarly one can define a group (or semigroup) generated by a noninvertible automaton. As an example we consider the semigroup of intermediate growth generated by the two state automaton ([BRS06])

```
gap> SG := AutomatonSemigroup( "f0=(f0,f0)(1,2), f1=(f1,f0)[2,2]" );
< f0, f1 >
```

Another type of groups (semigroups) implemented in the package is the class of groups (semigroups) defined by wreath recursion (finitely generated self-similar groups).

```
gap> WRG := SelfSimilarGroup("x=(1,y)(1,2),y=(z^-1,1)(1,2),z=(1,x*y)");
< x, y, z >
```

Now we can compute several properties of `Grigorchuk_Group`, `Basilica` and `SG`

```
gap> IsFinite(Grigorchuk_Group);
false
gap> IsSphericallyTransitive(Grigorchuk_Group);
true
gap> IsFractal(Grigorchuk_Group);
true
gap> IsAbelian(Grigorchuk_Group);
false
gap> IsTransitiveOnLevel(Grigorchuk_Group, 4);
true
```

We can also check that `Basilica` and `WRG` are contracting and compute their nuclei

```
gap> IsContracting(Basilica);
true
gap> GroupNucleus(Basilica);
[ 1, u, v, u^-1, v^-1, u^-1*v, v^-1*u ]
gap> IsContracting( WRG );
true
gap> GroupNucleus( WRG );
[ 1, y*z^-1*x*y, z^-1*y^-1*x^-1*y*z^-1, z^-1*y^-1*x^-1, y^-1*x^-1*z*y^-1,
  z*y^-1*x*y*z, x*y*z ]
```

The group `Grigorchuk_Group` is generated by a bounded automaton and, thus, is amenable (see [BKN10])

```
gap> IsGeneratedByBoundedAutomaton(Grigorchuk_Group);
true
gap> IsAmenable(Grigorchuk_Group);
true
```

We can compute the stabilizers of levels and vertices

```
gap> StabilizerOfLevel(Grigorchuk_Group, 2);
< a*b*a*d*a^-1*b^-1*a^-1, d, b*a*d*a^-1*b^-1, a*b*c*a^-1, b*a*b*a*b^-1*a^-1*b^
-1*a^-1, a*b*a*b*a*b^-1*a^-1*b^-1 >
gap> StabilizerOfVertex(Grigorchuk_Group, [2, 1]);
< a*b*a*d*a^-1*b^-1*a^-1, d, a*c*b^-1*a^-1, c, b, a*b*a*c*a^-1*b^-1*a^
-1, a*b*a*b*a^-1*b^-1*a^-1 >
```

In the case of a finite group we can produce an isomorphism into a permutation group

```
gap> f := IsomorphismPermGroup(Group(a,b));
[ a, b ] ->
[ (1,2)(3,5)(4,6)(7,9)(8,10)(11,13)(12,14)(15,17)(16,18)(19,21)(20,22)(23,
    25)(24,26)(27,29)(28,30)(31,32), (1,3)(2,4)(5,7)(6,8)(9,11)(10,12)(13,
    15)(14,16)(17,19)(18,20)(21,23)(22,24)(25,27)(26,28)(29,31)(30,32) ]
gap> Size(Image(f));
32
```

Here is how to find relations in `Basilica` between elements of length not greater than 5.

```
gap> FindGroupRelations(Basilica, 6);
v*u*v*u^-1*v^-1*u*v^-1*u^-1
v*u^2*v^-1*u^2*v*u^-2*v^-1*u^-2
v^2*u*v^2*u^-1*v^-2*u*v^-2*u^-1
[ v*u*v*u^-1*v^-1*u*v^-1*u^-1, v*u^2*v^-1*u^2*v*u^-2*v^-1*u^-2,
  v^2*u*v^2*u^-1*v^-2*u*v^-2*u^-1 ]
```

Or relations in the subgroup $\langle p = uv^{-1}, q = vu \rangle$

```
gap> FindGroupRelations([u*v^-1,v*u], ["p", "q"], 5);
q*p^2*q*p^-1*q^-2*p^-1
[ q*p^2*q*p^-1*q^-2*p^-1 ]
```

Or relations in the semigroup `SG`

```
gap> FindSemigroupRelations(SG, 4);
f0^3 = f0
f0^2*f1 = f1
f1*f0^2 = f1
f1^3 = f1
[ [ f0^3, f0 ], [ f0^2*f1, f1 ], [ f1*f0^2, f1 ], [ f1^3, f1 ] ]
```

Some basic operations with elements are the following:

The function `IsOne` computes whether an element represents the trivial automorphism of the tree

```
gap> IsOne( (a*b)^16 );
true
```

Here is how to compute the order (this function might not stop in some cases)

```
gap> Order(a*b);
16
gap> Order(u^22*v^-15*u^2*v*u^10);
infinity
```

Note that the package contains many helpful notifications that can be enabled by changing `InfoLevel` for `InfoAutomGrp`. In many situations level 3 provides additional information about the computation that can

be used either to track the progress or to extract the proof from the result as it can be done in the example below.

```
gap> SetInfoLevel(InfoAutomGrp,3);
gap> Order(u^22*v^-15*u^2*v*u^10);
#I  v is obtained from (u^22*v^-15*u^2*v*u^10)^1
    by taking sections and cyclic reductions at vertex [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
#I  v is obtained from (v)^2
    by taking sections and cyclic reductions at vertex [ 1, 1 ]
infinity
```

One can check if a particular element acts spherically transitively on the tree (this function might not stop in some cases)

```
gap> IsSphericallyTransitive(a*b);
false
gap> IsSphericallyTransitive(u*v);
true
```

The sections of an element can be obtained as follows

```
gap> Section(u*v^2*u, 2);
u^2*v
gap> Decompose(u*v^2*u);
(v, u^2*v)
gap> Decompose(u*v^2*u, 3);
(v, 1, 1, 1, u*v, 1, u, 1)(1,2)(5,6)
```

One can try to compute whether the elements of group `WRG` defined by wreath recursion are finite-state and calculate corresponding automaton

```
gap> IsFiniteState(x*y^-1);
true
gap> AllSections(x*y^-1);
[ x*y^-1, z, 1, x*y, y*z^-1, z^-1*y^-1*x^-1, y^-1*x^-1*z*y^-1, z*y^-1*x*y*z,
  y*z^-1*x*y, z^-1*y^-1*x^-1*y*z^-1, x*y*z, y, z^-1, y^-1*x^-1, z*y^-1 ]
gap> A := MealyAutomaton(x*y^-1);
<automaton>
gap> NumberOfStates(A);
15
```

To get the action of an element on a vertex or on a particular level of the tree use the following commands

```
gap> [1,2,1,1]^(a*b);
[ 2, 2, 1, 1 ]
gap> PermOnLevel(u*v^2*v, 3);
(1,6,4,8,2,5,3,7)
```

The action of the whole group `Grigorchuk_Group` on some level can be computed via `PermGroupOnLevel` (see 2.3.1).

```
gap> PermGroupOnLevel(Grigorchuk_Group, 3);
Group([ (1,5)(2,6)(3,7)(4,8), (1,3)(2,4)(5,6), (1,3)(2,4), (5,6) ])
gap> Size(last);
128
```

The next example shows how to find all elements of length at most 5 of order 16 in the Grigorchuk group.

```
gap> FindElements(Grigorchuk_Group, Order, 16, 5);
[ a*b, b*a, c*a*d, d*a*c, a*b*a*d, a*c*a*d, a*d*a*b, a*d*a*c, b*a*d*a,
  c*a*d*a, d*a*b*a, d*a*c*a, a*c*a*d*a, a*d*a*c*a, b*a*b*a*c, b*a*c*a*c,
  c*a*b*a*b, c*a*c*a*b ]
```

2 Properties and operations with groups and semigroups

In this chapter we present the functionality applicable to groups and semigroups.

2.1 Creation of groups and semigroups

1 ▶ `AutomatonGroup(` *string*`[,` *bind_vars*`] )` O
▶ `AutomatonGroup(` *list*`[,` *names*`,` *bind_vars*`] )` O
▶ `AutomatonGroup(` *automaton*`[,` *bind_vars*`] )` O

Creates the self-similar group generated by the finite automaton, described by *string* or *list*, or by the argument *automaton*.

The argument *string* is a conventional notation of the form `name1=(name11,name12,...,name1d)perm1, name2=...` where each `name*` is a name of a state or `1`, and each `perm*` is a permutation written in GAP notation. Trivial permutations may be omitted. This function ignores whitespace, and states may be separated by commas or semicolons.

The argument *list* is a list consisting of n entries corresponding to n states of an automaton. Each entry is of the form $[a_1, ..., a_d, p]$, where $d \geq 2$ is the size of the alphabet the group acts on, a_i are `IsInt` in $\{1, \ldots, n\}$ and represent the sections of the corresponding state at all vertices of the first level of the tree; and p from `SymmetricGroup(`d`)` describes the action of the corresponding state on the alphabet.

The optional argument *names* must be a list of names of generators of the group, corresponding to the states of the automaton. These names are used to display elements of the resulting group.

If the optional argument *bind_vars* is `false` the names of generators of the group are not assigned to the global variables. The default value is `true`. One can use `AssignGeneratorVariables` function to assign these names later, if they were not assigned when the group was created.

```
gap> AutomatonGroup("a=(a,b), b=(a, b)(1,2)");
< a, b >
gap> AutomatonGroup("a=(b,a,1)(2,3), b=(1,a,b)(1,2,3)");
< a, b >
gap> A := MealyAutomaton("a=(b,1)(1,2), b=(a,1)");
<automaton>
gap> G := AutomatonGroup(A);
< a, b >
```

In the second form of this operation the definition of the first group looks like

```
gap> AutomatonGroup([ [ 1, 2, ()], [ 1, 2, (1,2) ] ], [ "a", "b" ]);
< a, b >
```

2 ▶ `AutomatonSemigroup(` *string*`[,` *bind_vars*`] )` O
▶ `AutomatonSemigroup(` *list*`[,` *names*`,` *bind_vars*`] )` O
▶ `AutomatonSemigroup(` *automaton*`[,` *bind_vars*`] )` O

Creates the semigroup generated by the finite automaton, described by *string* or *list*, or by the argument *automaton*.

The argument *string* is a conventional notation of the form `name1=(name11,name12,...,name1d)trans1, name2=...` where each `name*` is a name of a state or `1`, and each `trans*` is either a permutation written in GAP notation, or a list defining a transformation of the alphabet via `Transformation(trans*)`. Trivial permutations may be omitted. This function ignores whitespace, and states may be separated by commas or semicolons.

The argument *list* is a list consisting of n entries corresponding to n states of the automaton. Each entry is of the form $[a_1, \cdots, a_d, p]$, where $d \geq 2$ is the size of the alphabet the group acts on, a_i are `IsInt` in $\{1, \ldots, n\}$ and represent the sections of the corresponding state at all vertices of the first level of the tree; and p is a transformation of the alphabet describing the action of the corresponding state on the alphabet.

The optional arguments *names* and *bind_vars* have the same meaning as in `AutomatonGroup` (see 2.1.1).

```
gap> AutomatonSemigroup("a=(a, b)[2,2], b=(a,b)(1,2)");
< a, b >
gap> AutomatonSemigroup("a=(b,a,1)[1,1,3], b=(1,a,b)(1,2,3)");
< 1, a, b >
gap> A := MealyAutomaton("f0=(f0,f0)(1,2), f1=(f1,f0)[2,2]");
<automaton>
gap> G := AutomatonSemigroup(A);
< f0, f1 >
```

In the second form of this operation the definition of the second semigroup looks like

```
gap> AutomatonSemigroup([ [1,2,Transformation([2,2])], [ 1,2,(1,2)] ], ["a","b"]);
< a, b >
```

3 ▶ `SelfSimilarGroup(` *string*`[,` *bind_vars*`] )` O
▶ `SelfSimilarGroup(` *list*`[,` *names*`,` *bind_vars*`] )` O
▶ `SelfSimilarGroup(` *automaton*`[,` *bind_vars*`] )` O

Creates the self-similar group generated by the wreath recursion, described by *string* or *list*, or given by the argument *automaton*.

The argument *string* is a conventional notation of the form `name1=(word11,word12,...,word1d)perm1, name2=...` where each `name*` is a name of a state, `word*` is an associative word over the alphabet consisting of all `name*`, and each `perm*` is a permutation written in GAP notation. Trivial permutations may be omitted. This function ignores whitespace, and states may be separated by commas or semicolons.

The argument *list* is a list consisting of n entries corresponding to n generators of the group. Each entry is of the form $[a_1, ..., a_d, p]$, where $d \geq 2$ is the size of the alphabet the group acts on, a_i are lists acceptable by `AssocWordByLetterRep` (e.g. if the names of generators are `x`, `y` and `z`, then `[1, 1, -2, -2, 1, 3]` will produce `x^2*y^-2*x*z`) representing the sections of the corresponding generator at all vertices of the first level of the tree; and p from `SymmetricGroup(`d`)` describes the action of the corresponding generator on the alphabet.

The optional argument *names* must be a list of names of generators of the group. These names are used to display the elements of the resulting group.

If the optional argument *bind_vars* is `false` the names of generators of the group are not assigned to the global variables. The default value is `true`. One can use `AssignGeneratorVariables` function to assign these names later, if they were not assigned when the group was created.

```
gap> SelfSimilarGroup("a=(a*b, b^-1), b=(1, b^2*a)(1,2)");
< a, b >
gap> SelfSimilarGroup("a=(b,a,a^-1)(2,3), b=(1,a*b,b)(1,2,3)");
< a, b >
gap> A := MealyAutomaton("f0=(f0,f0)(1,2),f1=(f1,f0)");
<automaton>
gap> SelfSimilarGroup(A);
< f0, f1 >
```

In the second form of this operation the definition of the first group looks like

```
gap> SelfSimilarGroup([[ [1,2], [-2], ()], [ [], [2,2,1], (1,2) ]], ["a","b"]);
< a, b >
```

4 ▶ `SelfSimilarSemigroup(` *string*`[,` *bind_vars*`] )` O
▶ `SelfSimilarSemigroup(` *list*`[,` *names*`,` *bind_vars*`] )` O
▶ `SelfSimilarSemigroup(` *automaton*`[,` *bind_vars*`] )` O

Creates the semigroup generated by the wreath recursion, described by *string* or *list*, or given by the argument *automaton*. Note, that on the contrary to `AutomatonSemigroup` (2.1.2) in some cases the defined semigroup may not be self-similar, since the sections of generators may include inverses of generators or trivial homomorphisms, not included in the semigroup generated by the generators. If one needs to have self-similarity it is always possible to include the necessary sections in the generating set.

The argument *string* is a conventional notation of the form `name1=(word11,word12,...,word1d)trans1, name2=...` where each `name*` is a name of a state, `word*` is an associative word over the alphabet consisting of all `name*`, and each `trans*` is either a permutation written in GAP notation, or a list defining a transformation of the alphabet via `Transformation(trans*)`. Trivial permutations may be omitted. This function ignores whitespace, and states may be separated by commas or semicolons.

The argument *list* is a list consisting of n entries corresponding to n generators of the semigroup. Each entry is of the form $[a_1, ..., a_d, p]$, where $d \geq 2$ is the size of the alphabet the semigroup acts on, a_i are lists acceptable by `AssocWordByLetterRep` (e.g. if the names of generators are `x`, `y` and `z`, then `[1, 1, 2, 3]` will produce `x^2*y*z`) representing the sections of the corresponding generator at all vertices of the first level of the tree; and p is a transformation of the alphabet describing the action of the corresponding generator.

The optional arguments *names* and *bind_vars* have the same meaning as in `SelfSimilarGroup` (see 2.1.3).

```
gap> SelfSimilarSemigroup("a=(a*b,b)[1,1], b=(a,b^2*a)(1,2)");
< a, b >
gap> SelfSimilarSemigroup("a=(b,a,a^3)(2,3), b=(1,a*b,b^-1)(1,2,3)");
< a, b >
gap> A := MealyAutomaton("f0=(f0,f0)(1,2), f1=(f1,f0)[2,2]");
<automaton>
gap> SelfSimilarSemigroup(A);
< f0, f1 >
```

In the second form of this operation the definition of the first semigroup looks like

```
gap> SelfSimilarSemigroup([[[1,2], [2], ()], [[1], [2,2,1], (1,2)]],["a","b"]);
< a, b >
```

5 ▶ `IsTreeAutomorphismGroup(` G `)` C

The category of groups of tree automorphisms.

6 ▶ `IsAutomGroup( G )` C

The category of groups generated by finite invertible initial automata (elements from category `IsAutom`).

7 ▶ `IsAutomatonGroup( G )` P

is `true` if G is created using the command `AutomatonGroup` (2.1.1) or if the generators of G coincide with the generators of the corresponding family, and `false` otherwise. To test whether G is self-similar use `IsSelfSimilar` (2.2.8) command.

8 ▶ `IsSelfSimGroup( G )` C

The category of groups whose generators are defined using wreath recursion (elements from category `IsSelfSim`). These groups need not be self-similar.

9 ▶ `IsSelfSimilarGroup( G )` P

is `true` if G is created using the command `SelfSimilarGroup` (2.1.3) or if the generators of G coincide with the generators of the corresponding family, and `false` otherwise. To test whether G is self-similar use `IsSelfSimilar` (2.2.8) command.

2.2 Basic properties of groups and semigroups

1 ▶ `TopDegreeOfTree( obj )` A

Returns the degree of the tree on the first level, i.e. the number of vertices adjacent to the root vertex.

2 ▶ `DegreeOfTree( obj )` A

This is a synonym for TopDegreeOfTree (2.2.1) for the case of a regular tree. It is an error to call this method for an object which acts on a non-regular tree.

3 ▶ `IsFractal( G )` P

Returns whether the group G is fractal (also called as *self-replicating*). In other words, if G acts transitively on the first level and for any vertex v of the tree the projection of the stabilizer of v in G on this vertex coincides with the whole group G.

```
gap> Grigorchuk_Group := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> IsFractal(Grigorchuk_Group);
true
```

4 ▶ `IsFractalByWords( G )` P

Computes the generators of stabilizers of vertices of the first level and their projections on these vertices. Returns `true` if the preimages of these projections in the free group under the canonical epimorphism generate the whole free group for each stabilizer, and the G acts transitively on the first level. This is sufficient but not necessary condition for G to be fractal. See also `IsFractal` (2.2.3).

5 ▶ `IsSphericallyTransitive( G )` P

Returns whether the group G is spherically transitive (see 1.1).

```
gap> Grigorchuk_Group := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> IsSphericallyTransitive(Grigorchuk_Group);
true
```

6 ▶ `ContainsSphericallyTransitiveElement( G )` A

For a self-similar group G acting on a binary tree returns `true` if G contains an element acting spherically transitively on the levels of the tree and `false` otherwise. See also `SphericallyTransitiveElement` (2.3.15).

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> ContainsSphericallyTransitiveElement(Basilica);
true
gap> G := SelfSimilarGroup("a=(a^-1*b^-1,1)(1,2), b=(b^-1,a*b)");
< a, b >
gap> ContainsSphericallyTransitiveElement(G);
false
```

7 ▶ `IsTransitiveOnLevel( G, lev )` O

Returns whether the group (semigroup) G acts transitively on level *lev*.

```
gap> Grigorchuk_Group := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> IsTransitiveOnLevel(Group([a,b]),3);
true
gap> IsTransitiveOnLevel(Group([a,b]),4);
false
```

8 ▶ `IsSelfSimilar( G )` P

Returns whether the group or semigroup G is self-similar (see 1.1).

9 ▶ `IsContracting( G )` A

Given a self-similar group G tries to compute whether it is contracting or not. Only a partial method is implemented (since there is no general algorithm so far). First it tries to find the nucleus up to size 50 using `FindNucleus`(G,50) (see 2.3.18), then it tries to find evidence that the group is noncontracting using `IsNoncontracting`(G,10,10) (see 2.2.10). If the answer was not found one can try to use `FindNucleus` and `IsNoncontracting` with bigger parameters. Also one can use `SetInfoLevel(InfoAutomGrp, 3)` for more information to be displayed.

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> IsContracting(Basilica);
true
gap> IsContracting(AutomatonGroup("a=(c,a)(1,2), b=(c,b), c=(b,a)"));
false
```

10 ▶ `IsNoncontracting( G[, max_len, depth] )` F

Tries to show that the group G is not contracting. Enumerates the elements of the group G up to length *max_len* until it finds an element which has a section g of infinite order, such that `OrderUsingSections`(g, *depth*) (see 3.2.6) returns `infinity` and such that g stabilizes some vertex and has itself as a section at this vertex. See also `IsContracting` (2.2.9).

If *max_len* and *depth* are omitted they are assumed to be `infinity` and 10, respectively.

If `InfoLevel` of `InfoAutomGrp` is greater than or equal to 3 (one can set it by `SetInfoLevel( InfoAutomGrp, 3)`), then the proof is printed.

```
gap> G := AutomatonGroup("a=(b,a)(1,2), b=(c,b), c=(c,a)");
< a, b, c >
gap> IsNoncontracting(G);
true
gap> H := AutomatonGroup("a=(c,b)(1,2), b=(b,a), c=(a,a)");
< a, b, c >
gap> SetInfoLevel(InfoAutomGrp, 3);
gap> IsNoncontracting(H);
#I  There are 37 elements of length up to 2
#I  There are 187 elements of length up to 3
#I  a^2*c^-1*b^-1 is obtained from (a^2*c^-1*b^-1)^2
    by taking sections and cyclic reductions at vertex [ 1, 1 ]
#I  a^2*c^-1*b^-1 has b*c*a^-2 as a section at vertex [ 2 ]
true
```

11 ▶ `IsGeneratedByAutomatonOfPolynomialGrowth( G )` P

For a group G generated by all states of a finite automaton (see 2.1.7) determines whether this automaton has polynomial growth in terms of Sidki [Sid00].

See also operations `IsGeneratedByBoundedAutomaton` (2.2.12) and `PolynomialDegreeOfGrowthOfUnderlyingAutomaton` (2.2.13).

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> IsGeneratedByAutomatonOfPolynomialGrowth(Basilica);
true
gap> D := AutomatonGroup( "a=(a,b)(1,2), b=(b,a)" );
< a, b >
gap> IsGeneratedByAutomatonOfPolynomialGrowth(D);
false
```

12 ▶ `IsGeneratedByBoundedAutomaton( G )` P

For a group G generated by all states of a finite automaton (see 2.1.7) determines whether this automaton is bounded in terms of Sidki [Sid00].

See also `IsGeneratedByAutomatonOfPolynomialGrowth` (2.2.11) and `PolynomialDegreeOfGrowthOfUnderlyingAutomaton` (2.2.13).

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> IsGeneratedByBoundedAutomaton(Basilica);
true
gap> C := AutomatonGroup("a=(a,b)(1,2), b=(b,c), c=(c,1)(1,2)");
< a, b, c >
gap> IsGeneratedByBoundedAutomaton(C);
false
```

13 ▶ `PolynomialDegreeOfGrowthOfUnderlyingAutomaton( G )` A

For a group G generated by all states of a finite automaton (see 2.1.7) of polynomial growth in terms of Sidki [Sid00] determines the degree of polynomial growth of this automaton. This degree is 0 if and only if the automaton is bounded. If the growth of automaton is exponential returns `fail`.

See also `IsGeneratedByAutomatonOfPolynomialGrowth` (2.2.11) and `IsGeneratedByBoundedAutomaton` (2.2.12).

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> PolynomialDegreeOfGrowthOfUnderlyingAutomaton(Basilica);
0
gap> C := AutomatonGroup("a=(a,b)(1,2), b=(b,c), c=(c,1)(1,2)");
< a, b, c >
gap> PolynomialDegreeOfGrowthOfUnderlyingAutomaton(C);
2
```

14 ▶ `IsOfSubexponentialGrowth(` *G*`[,` *len*`,` *depth*`] )` O

Tries to check whether the growth function of a self-similar group G is subexponential. The main part of the algorithm works as follows. It looks at all words of length up to *len* and if for some length l for each word of this length l the sum of the lengths of all its sections at level *depth* is less then l, returns `true`. The default values of *len* and *depth* are 10 and 6 respectively. Setting `SetInfoLevel(InfoAtomGrp, 3)` will make it print for each length the words that are not contracted. It also sometimes helps to use `AG_UseRewritingSystem` (see 2.6.1).

```
gap> Grigorchuk_Group := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> AG_UseRewritingSystem(Grigorchuk_Group);
gap> IsOfSubexponentialGrowth(Grigorchuk_Group,10,6);
true
```

15 ▶ `IsAmenable(` *G* `)` P

In certain cases (for groups generated by bounded automata [BKN10], some virtually abelian groups or finite groups) returns `true` if G is amenable.

```
gap> Grigorchuk_Group := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> IsAmenable(Grigorchuk_Group);
true
```

16 ▶ `UnderlyingAutomaton(` *G* `)` A

For a group (or semigroup) G returns an automaton generating a self-similar group (or semigroup) containing G.

```
gap> GS := AutomatonSemigroup("x=(x,y)[1,1], y=(y,y)(1,2)");
< x, y >
gap> A := UnderlyingAutomaton(GS);
<automaton>
gap> Display(A);
a1 = (a1, a2)[ 1, 1 ], a2 = (a2, a2)[ 2, 1 ]
```

For a subgroup of Basilica group we get the automaton generating Basilica group.

```
gap> H := Group([u*v^-1,v^2]);
< u*v^-1, v^2 >
gap> Display(UnderlyingAutomaton(H));
a1 = (a1, a1), a2 = (a3, a1)(1,2), a3 = (a2, a1)
```

17 ▶ `AutomatonList(` *G* `)` AM

Returns an `AutomatonList` of `UnderlyingAutomaton`(G) (see 2.2.16).

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> AutomatonList(Basilica);
[ [ 2, 5, (1,2) ], [ 1, 5, () ], [ 5, 4, (1,2) ], [ 3, 5, () ], [ 5, 5, () ] ]
```

18 ▶ `RecurList( G )` AM

Returns an internal representation of the wreath recursion of the self-similar group (semigroup) containing G.

```
gap> R := SelfSimilarGroup("a=(a^-1*b,b^-1*a)(1,2), b=(a^-1,b^-1)");
< a, b >
gap> RecurList(R);
[ [ [ -1, 2 ], [ -2, 1 ], (1,2) ], [ [ -1 ], [ -2 ], () ],
  [ [ -1, 2 ], [ -2, 1 ], (1,2) ], [ [ 1 ], [ 2 ], () ] ]
```

2.3 Operations with groups and semigroups

1 ▶ `PermGroupOnLevel( G, k )` O

Returns the group of permutations induced by the action of the group G at the k-th level.

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> PermGroupOnLevel(Basilica, 4);
Group([ (1,11,3,9)(2,12,4,10)(5,13)(6,14)(7,15)(8,16), (1,6,2,5)(3,7)(4,8) ])
gap> H := PermGroupOnLevel(Group([u,v^2]),4);
Group([ (1,11,3,9)(2,12,4,10)(5,13)(6,14)(7,15)(8,16), (1,2)(5,6) ])
gap> Size(H);
64
```

2 ▶ `TransformationSemigroupOnLevel( G, k )` O

Returns the semigroup of transformations induced by the action of the semigroup G at the k-th level.

```
gap> S := AutomatonSemigroup("y=(1,u)[1,1],u=(y,u)(1,2)");
< 1, y, u >
gap> T := TransformationSemigroupOnLevel(S,3);
<transformation monoid on 8 pts with 2 generators>
gap> Size(T);
11
```

3 ▶ `StabilizerOfLevel( G, k )` O

Returns the stabilizer of the k-th level.

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> StabilizerOfLevel(Basilica, 2);
< u^2, v^2, u*v^2*u^-1, v*u^2*v^-1, u*v*u^2*v^-1*u^-1, (v*u)^2*(v^-1*u^-1)^2, v*u*\
v^2*u^-1*v^-1, (u*v)^2*u*v^-1*u^-1*v^-1, (u*v)^2*v*u^-1*v^-1*u^-1 >
```

4 ▶ `StabilizerOfFirstLevel( G )` A

Returns the stabilizer of the first level, see also 2.3.3.

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> StabilizerOfFirstLevel(Basilica);
< v, u^2, u*v*u^-1 >
```

5 ▶ `StabilizerOfVertex(` *G*, *v* `)` O

Returns the stabilizer of the vertex v. Here v can be a list representing a vertex, or a positive integer representing a vertex at the first level.

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> StabilizerOfVertex(Basilica, [1,2,1]);
< u^2, u*v*u^-1, v^2, v*u*v*u^-1*v^-1, v*u^-1*v*u*v^-1, v*u^4*v^-1, v*u^2*v^2*u^-2\
*v^-1, (v*u^2)^2*v^-1*u^-2*v^-1, v*u*(u*v)^2*u^-1*v^-1*u^-2*v^-1 >
```

6 ▶ `FixesLevel(` *obj*, *lev* `)` O

Returns whether *obj* fixes level *lev*, i.e. fixes every vertex at the level *lev*.

7 ▶ `FixesVertex(` *obj*, *v* `)` O

Returns whether *obj* fixes the vertex v. The vertex v may be given as a list, or as a positive integer, in which case it denotes the v-th vertex at the first level.

8 ▶ `Projection(` *G*, *v* `)` O
▶ `ProjectionNC(` *G*, *v* `)` O

Returns the projection of the group G at the vertex v. The group G must fix the vertex v, otherwise `Error`() will be called. The operation `ProjectionNC` does the same thing, except it does not check whether G fixes the vertex v.

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> Projection(StabilizerOfVertex(Basilica, [1,2,1]), [1,2,1]);
< u, v >
```

9 ▶ `ProjStab(` *G*, *v* `)` O

Returns the projection of the stabilizer of v at itself. It is a shortcut for `Projection`(`StabilizerOfVertex`(G, v), v) (see 2.3.8, 2.3.5).

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> ProjStab(Basilica, [1,2,1]);
< u, v >
```

10 ▶ `FindGroupRelations(` *G*`[,` *max_len*, *max_num_rels*`]` `)` O
▶ `FindGroupRelations(` *subs_words*`[,` *names*, *max_len*, *max_num_rels*`]` `)` O

Finds group relations between the generators of the group G or in the group generated by *subs_words*. Stops after investigating all words of length up to *max_len* elements or when it finds *max_num_rels* relations. The optional argument *names* is a list of names of generators of the same length as *subs_words*. If this argument is given the relations are given in terms of these names. Otherwise they are given in terms of the elements of the group generated by *subs_words*. If *max_len* or *max_num_rels* are not specified, they are assumed to be `infinity`. Note that if the rewring system (see 2.6.1) for group G is used, then this operation returns relations not contained in the rewriting system rules (see 2.6.4). This operation can be applied to any group, not only to a group generated by automata.

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> FindGroupRelations(Basilica, 6);
v*u*v*u^-1*v^-1*u*v^-1*u^-1
v*u^2*v^-1*u^2*v*u^-2*v^-1*u^-2
v^2*u*v^2*u^-1*v^-2*u*v^-2*u^-1
[ v*u*v*u^-1*v^-1*u*v^-1*u^-1, v*u^2*v^-1*u^2*v*u^-2*v^-1*u^-2,
  v^2*u*v^2*u^-1*v^-2*u*v^-2*u^-1 ]
gap> FindGroupRelations([u*v^-1, v*u], ["x", "y"], 5);
y*x^2*y*x^-1*y^-2*x^-1
[ y*x^2*y*x^-1*y^-2*x^-1 ]
gap> FindGroupRelations([u*v^-1, v*u], 5);
u^-2*v*u^-2*v^-1*u^2*v*u^2*v^-1
[ u^-2*v*u^-2*v^-1*u^2*v*u^2*v^-1 ]
gap> FindGroupRelations([(1,2)(3,4), (1,2,3)], ["x", "y"]);
x^2
y^-3
(y^-1*x)^3
[ x^2, y^-3, (y^-1*x)^3 ]
```

11 ▶ `FindSemigroupRelations(` *G*`[,` *max_len*`,` *max_num_rels*`] )` O

▶ `FindSemigroupRelations(` *subs_words*`[,` *names*`,` *max_len*`,` *max_num_rels*`] )` O

Finds semigroup relations between the generators of the group or semigroup G, or in the semigroup generated by *subs_words*. The arguments have the same meaning as in `FindGroupRelations` (2.3.10). It returns a list of pairs of equal words. In order to make the list of relations shorter it also tries to remove relations that can be derived from the known ones. Note, that by default the trivial automorphism is not included in every semigroup. So if one needs to find relations of the form $w = 1$ one has to define G as a monoid or to include the trivial automorphism into *subs_words* (for instance, as `One(g)` for any element `g` acting on the same tree). This operation can be applied for any semigroup, not only for a semigroup generated by automata.

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> FindSemigroupRelations([u*v^-1, v*u], ["x", "y"], 6);
y*x^2*y=x*y^2*x
y*x^3*y^2=x^2*y^3*x
y^2*x^3*y=x*y^3*x^2
[ [ y*x^2*y, x*y^2*x ], [ y*x^3*y^2, x^2*y^3*x ], [ y^2*x^3*y, x*y^3*x^2 ] ]
gap> FindSemigroupRelations([u*v^-1, v*u],6);
v*u^2*v^-1*u^2 = u^2*v*u^2*v^-1
v*u*(u*v^-1)^2*u^2*v*u = u*v^-1*u*(u*v)^2*u^2*v^-1
(v*u)^2*(u*v^-1)^2*u^2 = u*(u*v)^2*u*(u*v^-1)^2
[ [ v*u^2*v^-1*u^2, u^2*v*u^2*v^-1 ],
  [ v*u*(u*v^-1)^2*u^2*v*u, u*v^-1*u*(u*v)^2*u^2*v^-1 ],
  [ (v*u)^2*(u*v^-1)^2*u^2, u*(u*v)^2*u*(u*v^-1)^2 ] ]
gap> x := Transformation([1,1,2]);;
gap> y := Transformation([2,2,3]);;
gap> FindSemigroupRelations([x,y],["x","y"]);
y*x=x
y^2=y
x^3=x^2
x^2*y=x*y
[ [ y*x, x ], [ y^2, y ], [ x^3, x^2 ], [ x^2*y, x*y ] ]
```

12 ▶ `Iterator( G[, max_len] )` M

Provides a possibility to loop over elements of a group (semigroup, monoid) generated by automata. If *max_len* is given, it stops after enumerating all elements of length up to *max_len*.

```
gap> Grigorchuk_Group := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> iter := Iterator(Grigorchuk_Group, 5);
<iterator>
gap> l:=[];;
gap> for g in iter do
>      if Order(g)=16 then Add(l,g); fi;
>    od;
gap> l;
[ b*a, a*b, d*a*c, c*a*d, d*a*c*a, d*a*b*a, c*a*d*a, b*a*d*a, a*d*a*c,
  a*d*a*b, a*c*a*d, a*b*a*d, c*a*c*a*b, c*a*b*a*b, b*a*c*a*c, b*a*b*a*c,
  a*d*a*c*a, a*c*a*d*a ]
```

13 ▶ `FindElement( G, func, val, max_len )` O
▶ `FindElements( G, func, val, max_len )` O

The first function enumerates elements of the group (semigroup, monoid) G until it finds an element g of length at most *max_len*, for which *func*(g)=*val*. Returns g if such an element was found and `fail` otherwise.

The second function enumerates elements of the group (semigroup, monoid) of length at most *max_len* and returns the list of elements g, for which *func*(g)=*val*.

These functions are based on `Iterator` operation (see 2.3.12), so can be applied in more general settings whenever GAP knows how to solve word problem in the group. The following example illustrates how to find an element of order 16 in Grigorchuk group and the list of all such elements of length at most 5.

```
gap> Grigorchuk_Group := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> FindElement(Grigorchuk_Group, Order, 16, 5);
a*b
gap> FindElements(Grigorchuk_Group,Order,16,5);
[ a*b, b*a, c*a*d, d*a*c, a*b*a*d, a*c*a*d, a*d*a*b, a*d*a*c, b*a*d*a, c*a*d*a,
  d*a*b*a, d*a*c*a, a*c*a*d*a, a*d*a*c*a, (b*a)^2*c, b*(a*c)^2, c*(a*b)^2,
  (c*a)^2*b ]
```

14 ▶ `FindElementOfInfiniteOrder( G, max_len, depth )` O
▶ `FindElementsOfInfiniteOrder( G, max_len, depth )` O

The first function enumerates elements of the group G up to length *max_len* until it finds an element g of infinite order, such that `OrderUsingSections`(g,*depth*) (see 3.2.6) is `infinity`. In other words all sections of every element up to depth *depth* are investigated. In case if the element belongs to the group generated by bounded automaton (see 2.2.12) one can set *depth* to be `infinity`.

The second function returns the list of all such elements up to length *max_len*.

```
gap> G := AutomatonGroup("a=(1,1)(1,2), b=(a,c), c=(b,1)");
< a, b, c >
gap> FindElementOfInfiniteOrder(G, 5, 10);
a*b*c
```

15 ▶ `SphericallyTransitiveElement( G )` A

For a self-similar group G acting on a binary tree returns an element of G acting spherically transitively on the levels of the tree if such an element exists and `fail` otherwise. See also `ContainsSphericallyTransitiveElement` (2.2.6).

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> SphericallyTransitiveElement(Basilica);
u*v
gap> G := SelfSimilarGroup("a=(a^-1*b^-1,1)(1,2), b=(b^-1,a*b)");
< a, b >
gap> SphericallyTransitiveElement(G);
fail
```

16 ▶ `Growth(` *G*`,` *max_len* `)` O

Returns a list of the first values of the growth function of a group (semigroup, monoid) G. If G is a monoid it computes the growth function at $\{0, 1, \ldots, max\_len\}$, and for a semigroup without identity at $\{1, \ldots, max\_len\}$.

```
gap> Grigorchuk_Group := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> Growth(Grigorchuk_Group, 7);
There are 11 elements of length up to 2
There are 23 elements of length up to 3
There are 40 elements of length up to 4
There are 68 elements of length up to 5
There are 108 elements of length up to 6
There are 176 elements of length up to 7
[ 1, 5, 11, 23, 40, 68, 108, 176 ]
gap> H := AutomatonSemigroup("a=(a,b)[1,1], b=(b,a)(1,2)");
< a, b >
gap> Growth(H,6);
[ 2, 6, 14, 30, 62, 126 ]
```

17 ▶ `ListOfElements(` *G*`,` *max_len* `)` O

Returns the list of all different elements of a group (semigroup, monoid) G up to length *max_len*.

```
gap> Grigorchuk_Group := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> ListOfElements(Grigorchuk_Group, 3);
[ 1, a, b, c, d, a*b, a*c, a*d, b*a, c*a, d*a, a*b*a, a*c*a, a*d*a, b*a*b,
  b*a*c, b*a*d, c*a*b, c*a*c, c*a*d, d*a*b, d*a*c, d*a*d ]
```

18 ▶ `FindNucleus(` *G*`[,` *max_nucl*`,` *print_info*`]` `)` O

Given a self-similar group G it tries to find its nucleus. If G is not contracting it will loop forever. When it finds the nucleus it returns the triple [`GroupNucleus`(G), `GeneratingSetWithNucleus`(G), `GeneratingSetWithNucleusAutom`(G)] (see 2.5.1, 2.5.2, 2.5.3).

If *max_nucl* is given it stops after finding *max_nucl* elements that need to be in the nucleus and returns `fail` if the nucleus was not found.

An optional argument *print_info* is a boolean telling whether to print results of intermediate computations. The default value is `true`.

Use `IsNoncontracting` (see 2.2.10) to try to show that G is noncontracting.

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> FindNucleus(Basilica);
Trying generating set with 5 elements
Elements added:[ u^-1*v, v^-1*u ]
Trying generating set with 7 elements
[ [ 1, u, v, u^-1, v^-1, u^-1*v, v^-1*u ],
  [ 1, u, v, u^-1, v^-1, u^-1*v, v^-1*u ], <automaton> ]
```

19 ▶ `LevelOfFaithfulAction( G )` A
▶ `LevelOfFaithfulAction( G, max_lev )` A

For a given finite self-similar group G determines the smallest level of the tree, where G acts faithfully, i.e. the stabilizer of this level in G is trivial. The idea here is that for a self-similar group all nontrivial level stabilizers are different. If *max_lev* is given it finds only first *max_lev* quotients by stabilizers and if all of them have different size it returns `fail`. If G is infinite and *max_lev* is not specified it will loop forever.

See also `IsomorphismPermGroup` (2.3.20).

```
gap> H := SelfSimilarGroup("a=(a,a)(1,2), b=(a,a), c=(b,a)(1,2)");
< a, b, c >
gap> LevelOfFaithfulAction(H);
3
gap> Size(H);
16
gap> Adding_Machine := AutomatonGroup("a=(1,a)(1,2)");
< a >
gap> LevelOfFaithfulAction(Adding_Machine, 10);
fail
```

20 ▶ `IsomorphismPermGroup( G )` O
▶ `IsomorphismPermGroup( G, max_lev )` O

For a given finite group G generated by initial automata or by elements defined by wreath recursion computes an isomorphism from G into a finite permutational group. If G is not known to be self-similar (see 2.2.8) the isomorphism is based on the regular representation, which works generally much slower. If G is self-similar there is a level of the tree (see 2.3.19), where G acts faithfully. The corresponding representation is returned in this case. If *max_lev* is given it finds only the first *max_lev* quotients by stabilizers and if all of them have different size it returns `fail`. If G is infinite and *max_lev* is not specified it will loop forever.

For example, consider a subgroup $\langle a, b\rangle$ of Grigorchuk group.

```
gap> Grigorchuk_Group := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> f := IsomorphismPermGroup(Group(a, b));
MappingByFunction( < a, b >, Group(
[ (1,2)(3,5)(4,6)(7,9)(8,10)(11,13)(12,14)(15,17)(16,18)(19,21)(20,22)(23,
    25)(24,26)(27,29)(28,30)(31,32), (1,3)(2,4)(5,7)(6,8)(9,11)(10,12)(13,
    15)(14,16)(17,19)(18,20)(21,23)(22,24)(25,27)(26,28)(29,31)(30,32)
 ]), function( g ) ... end, function( b ) ... end )
gap> Size(Image(f));
32
gap> H := SelfSimilarGroup("a=(a*b,1)(1,2), b=(1,b*a^-1)(1,2), c=(b, a*b)");
< a, b, c >
gap> f1 := IsomorphismPermGroup(H);
MappingByFunction( < a, b, c >, Group([ (1,3)(2,4), (1,3)(2,4), (1,2)
```

```
 ]), function( g ) ... end, function( b ) ... end )
gap> Size(Image(f1));
8
gap> PreImagesRepresentative(f1, (1,3,2,4));
a*c
gap> (a*c)^f1;
(1,3,2,4)
```

21 ▶ `Random( G )` O

Returns a random element of a group (semigroup) G. The operation is based on the generator of random elements in free groups and semigroups.

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> Random( Basilica );
v*u^-3
```

22 ▶ `MarkovOperator( G, lev, weights )` O

Computes the matrix of the Markov operator related to the (semi)group G on the *lev*-th level of the tree. If G is a group generated by $g_1, g_2, \ldots, g_n$, then the Markov operator is defined as (`PermOnLevelAsMatrix`$(g_1) + \cdots +$ `PermOnLevelAsMatrix`$(g_n) +$ `PermOnLevelAsMatrix`$(g_1^{-1}) + \cdots +$ `PermOnLevelAsMatrix`$(g_n^{-1}))/(2 * n)$. If S is a semigroup generated by $s_1, s_2, \ldots, s_n$, then the Markov operator is defined similarly with `PermOnLevelAsMatrix` being replaced with `TransformationOnLevelAsMatrix`. If the list of *weights* is given, uses its entries as coefficients of operators correspondings to the generators of a group or semigroup. In the case of a group, the length of *weights* must be twice as big as the number of generators of G. The list *weights* may consist either of numbers or of strings representing the names of indeterminates. See also `PermOnLevelAsMatrix` (3.2.9) and `TransformationOnLevelAsMatrix` (3.2.11).

```
gap> L := AutomatonGroup("p=(p,q)(1,2), q=(p,q)");
< p, q >
gap> MarkovOperator(L, 3);
[ [ 0, 0, 1/4, 1/4, 0, 1/4, 0, 1/4 ], [ 0, 0, 1/4, 1/4, 1/4, 0, 1/4, 0 ],
  [ 1/4, 1/4, 0, 0, 1/4, 0, 1/4, 0 ], [ 1/4, 1/4, 0, 0, 0, 1/4, 0, 1/4 ],
  [ 0, 1/4, 1/4, 0, 0, 1/2, 0, 0 ], [ 1/4, 0, 0, 1/4, 1/2, 0, 0, 0 ],
  [ 0, 1/4, 1/4, 0, 0, 0, 1/2, 0 ], [ 1/4, 0, 0, 1/4, 0, 0, 0, 1/2 ] ]
gap> MarkovOperator(L,3,["a","b","c","d"]);
[ [ 0, 0, d, b, 0, c, 0, a ], [ 0, 0, b, d, c, 0, a, 0 ],
  [ b, d, 0, 0, a, 0, c, 0 ], [ d, b, 0, 0, 0, a, 0, c ],
  [ 0, a, c, 0, 0, b+d, 0, 0 ], [ a, 0, 0, c, b+d, 0, 0, 0 ],
  [ 0, c, a, 0, 0, 0, b+d, 0 ], [ c, 0, 0, a, 0, 0, 0, b+d ] ]
```

In the case of semigroups we have:

```
gap> S := AutomatonSemigroup("c=(c,d)[1,1],d=(c,c)(1,2)");
< c, d >
gap> MarkovOperator(S,3,["w1","w2"]);
[ [ w1, 0, 0, 0, w2, 0, 0, 0 ], [ w1, 0, 0, 0, w2, 0, 0, 0 ],
  [ 0, w1, 0, 0, 0, w2, 0, 0 ], [ w1, 0, 0, 0, w2, 0, 0, 0 ],
  [ w2, 0, w1, 0, 0, 0, 0, 0 ], [ w2, 0, w1, 0, 0, 0, 0, 0 ],
  [ w1, w2, 0, 0, 0, 0, 0, 0 ], [ w1+w2, 0, 0, 0, 0, 0, 0, 0 ] ]
gap> MarkovOperator(S,3,[1/3,2/3]);
[ [ 1/3, 0, 0, 0, 2/3, 0, 0, 0 ], [ 1/3, 0, 0, 0, 2/3, 0, 0, 0 ],
  [ 0, 1/3, 0, 0, 0, 2/3, 0, 0 ], [ 1/3, 0, 0, 0, 2/3, 0, 0, 0 ],
```

```
[ 2/3, 0, 1/3, 0, 0, 0, 0, 0 ], [ 2/3, 0, 1/3, 0, 0, 0, 0, 0 ],
[ 1/3, 2/3, 0, 0, 0, 0, 0, 0 ], [ 1, 0, 0, 0, 0, 0, 0, 0 ] ]
```

23 ▶ `MihailovaSystem( G )` AM

In the case when G is an automaton fractal group acting on a binary tree, computes the generating set for the first level stabilizer in G such that the sections of these generators at the first level, viewed as elements of $F_r \times F_r$, are in Mihailova normal form. See [GS14] for details.

```
gap> G := AutomatonGroup("a=(b,c)(1,2),b=(a,c),c=(a,a)");
< a, b, c >
gap> M := MihailovaSystem(G);
[ c^-1*b, c^-1*b^-1*c*a^-1*b*c*b^-1*a, a^-1*b*c*b^-1*a, a*c^-1*b^-1*a*c,
  c^-1*a^-1*b*c*a ]
gap> for g in M do
>      Print(g,"=",Decompose(g),"\n");
>    od;
c^-1*b=(1, a^-1*c)
c^-1*b^-1*c*a^-1*b*c*b^-1*a=(1, a^-1*c^-1*a*b^-1*a*b)
a^-1*b*c*b^-1*a=(a, b^-1*a*b)
a*c^-1*b^-1*a*c=(b, c*a^-2*b*a)
c^-1*a^-1*b*c*a=(c, a^-1*b^-1*a^2*b)
```

24 ▶ `AbelImage( obj )` A

Returns image of *obj* in the canonical projection onto the abelianization of the full group of tree automorphisms, represented as a subgroup of the additive group of rational functions.

25 ▶ `DiagonalPower( fam[, k] )` O

For a given automaton group G acting on alphabet X and corresponding family *fam* of automata one can consider the action of G^k on X^k defined by $(x_1, x_2, \ldots, x_k)^{(g_1,g_2,\ldots,g_k)} = (x_1^{g_1}, x_2^{g_2}, \ldots, x_k^{g_k})$. This function constructs a self-similar group, which encodes this action. If k is not given it is assumed to be 2.

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> S := DiagonalPower(UnderlyingAutomFamily(Basilica));
< uu, uv, u1, vu, vv, v1, 1u, 1v >
gap> Decompose(uu);
(vv, v1, 1v, 1)(1,4)(2,3)
```

26 ▶ `MultAutomAlphabet( fam )` O

27 ▶ `UnderlyingAutomFamily( G )` A

Returns the family to which the elements of G belong.

2.4 Self-similar groups and semigroups defined by the wreath recursion

1 ▶ `IsFiniteState( G )` P

For a group or semigroup of homomorphisms of the tree defined using a wreath recursion, returns `true` if all generators can be represented as finite automata (have finitely many different sections). It will never stop if the free reduction of words is not sufficient to establish the finite-state property or if the group is not finite-state. In case G is a finite-state group it automatically computes the attributes `UnderlyingAutomatonGroup`(G) (2.4.4), `IsomorphicAutomGroup`(G) (2.4.2) and `MonomorphismToAutomatonGroup`(G) (2.4.6).

For a finite-state semigroup it computes the corresponding attributes `UnderlyingAutomatonSemigroup`(G) (2.4.5), `IsomorphicAutomSemigroup`(G) (2.4.3) and `MonomorphismToAutomatonSemigroup`(G) (2.4.7).

```
gap> W := SelfSimilarGroup("x=(x^-1,y)(1,2), y=(z^-1,1)(1,2), z=(1,x*y)");
< x, y, z >
gap> IsFiniteState(W);
true
gap> Size(GeneratorsOfGroup(UnderlyingAutomatonGroup(W)));
50
```

2 ▶ `IsomorphicAutomGroup( G )` AM

In case G is finite-state tries to compute a group generated by automata, isomorphic to G, which is a subgroup of `UnderlyingAutomatonGroup`(G) (see 2.4.4). The natural isomorphism between G and `IsomorphicAutomGroup`(G) is stored in the attribute `MonomorphismToAutomatonGroup`(G) (2.4.6). In some cases it may be useful to check if G is finite.

```
gap> R := SelfSimilarGroup("a=(a^-1*b,b^-1*a)(1,2), b=(a^-1,b^-1)");
< a, b >
gap> UR := UnderlyingAutomatonGroup(R);
< a1, a2, a4, a5 >
gap> IR := IsomorphicAutomGroup(R);
< a1, a5 >
gap> hom := MonomorphismToAutomatonGroup(R);
MappingByFunction( < a, b >, < a1, a5 >, function( a ) ... end, function( b ) \
... end )
gap> (a*b)^hom;
a1*a5
gap> PreImagesRepresentative(hom, last);
a*b
gap> List(GeneratorsOfGroup(UR), x -> PreImagesRepresentative(hom, x));
[ a, a^-1*b, b^-1*a, b ]
```

All these operations work also for the subgroups of groups generated by `SelfSimilarGroup`. (2.1.3).

```
gap> T := Group([b*a, a*b]);
< b*a, a*b >
gap> IT := IsomorphicAutomGroup(T);
< a1, a4 >
```

Note, that different groups have different `UnderlyingAutomGroup` attributes. For example, the generator `a1` of group `IT` above is different from the generator `a1` of group `IR`.

3 ▶ `IsomorphicAutomSemigroup( G )` AM

In case G is finite-state returns a semigroup generated by automata, isomorphic to G, which is a subsemigroup of `UnderlyingAutomatonSemigroup`(G) (see 2.4.5). The natural isomorphism between G and `IsomorphicAutomSemigroup`(G) is stored in the attribute `MonomorphismToAutomatonSemigroup`(G) (2.4.7).

```
gap> R := SelfSimilarSemigroup("a=(1,1)[1,1], b=(a*c,1)(1,2), c=(1,a*b)");
< a, b, c >
gap> UR := UnderlyingAutomatonSemigroup(R);
< 1, a1, a3, a5, a6 >
gap> IR := IsomorphicAutomSemigroup(R);
< a1, a3, a5 >
gap> hom := MonomorphismToAutomatonSemigroup(R);
```

```
MappingByFunction( < a, b, c >, < a1, a3, a5 >, function( a ) ... end, functio\
n( b ) ... end )
gap> (a*b)^hom;
a1*a3
gap> PreImagesRepresentative(hom, last);
a*b
gap> List(GeneratorsOfSemigroup(UR), x -> PreImagesRepresentative(hom, x));
[ 1, a, b, c, a*b ]
```

All these operations work also for the subsemigroups of semigroups generated by `SelfSimilarSemigroup` (2.1.4).

```
gap> T := Semigroup([a*b, b^2]);
< a*b, b^2 >
gap> IT := IsomorphicAutomSemigroup(T);
< a1, a4 >
```

Note, that different semigroups have different `UnderlyingAutomSemigroup` attributes. For example, the generator `a1` of semigroup `IT` above is different from the generator `a1` of semigroup `IR`.

4 ▶ `UnderlyingAutomatonGroup( G )` AM

In case G is finite-state returns a self-similar closure of G as a group generated by automaton. The natural monomorphism from G and `UnderlyingAutomatonGroup`(G) is stored in the attribute `MonomorphismToAutomatonGroup`(G) (2.4.6). If G is created by `SelfSimilarGroup` (see 2.1.3), then the self-similar closure of G coincides with G, so one can use `MonomorphismToAutomatonGroup`(G) to get preimages of elements of `UnderlyingAutomatonGroup`(G) in G. See the example for `IsomorphicAutomGroup` (2.4.2).

5 ▶ `UnderlyingAutomatonSemigroup( G )` AM

In case G is finite-state returns a self-similar closure of G as a semigroup generated by automaton. The natural monomorphism from G and `UnderlyingAutomatonSemigroup`(G) is stored in the attribute `MonomorphismToAutomatonSemigroup`(G) (2.4.7). If G is created by `SelfSimilarSemigroup` (see 2.1.4), then the self-similar closure of G coincides with G, so one can use `MonomorphismToAutomatonSemigroup`(G) to get preimages of elements of `UnderlyingAutomatonSemigroup`(G) in G. See the example for `IsomorphicAutomSemigroup` (2.4.3).

6 ▶ `MonomorphismToAutomatonGroup( G )` AM

In case G is finite-state returns a monomorphism from G into `UnderlyingAutomatonGroup`(G) (see 2.4.4). If G is created by `SelfSimilarGroup` (see 2.1.3), then one can use `MonomorphismToAutomatonGroup`(G) to get preimages of elements of `UnderlyingAutomatonGroup`(G) in G. See the example for `IsomorphicAutomGroup` (2.4.2).

7 ▶ `MonomorphismToAutomatonSemigroup( G )` AM

In case G is finite-state returns a monomorphism from G into `UnderlyingAutomatonSemigroup`(G) (see 2.4.5). If G is created by `SelfSimilarSemigroup` (see 2.1.4), then one can use `MonomorphismToAutomatonSemigroup`(G) to get preimages of elements of `UnderlyingAutomatonSemigroup`(G) in G. See the example for `IsomorphicAutomSemigroup` (2.4.3).

2.5 Contracting groups

1 ▶ `GroupNucleus( G )` AM

Tries to compute the *nucleus* (see the definition in 1.1) of a self-similar group G. Note that this set need not contain the original generators of G. It uses `FindNucleus` (see 2.3.18) operation and behaves accordingly: if the group is not contracting it will loop forever. See also `GeneratingSetWithNucleus` (2.5.2).

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> GroupNucleus(Basilica);
[ 1, u, v, u^-1, v^-1, u^-1*v, v^-1*u ]
```

2 ▶ `GeneratingSetWithNucleus( G )` AM

Tries to compute the generating set of a self-similar group G that includes the original generators and the *nucleus* (see 1.1) of G. It uses `FindNucleus` operation and behaves accordingly: if the group is not contracting it will loop forever (modulo memory constraints, of course). See also `GroupNucleus` (2.5.1).

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> GeneratingSetWithNucleus(Basilica);
[ 1, u, v, u^-1, v^-1, u^-1*v, v^-1*u ]
```

3 ▶ `GeneratingSetWithNucleusAutom( G )` AM

Computes the automaton of the generating set that includes the nucleus of a contracting group G. See also `GeneratingSetWithNucleus` (2.5.2).

```
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> B_autom := GeneratingSetWithNucleusAutom(Basilica);
<automaton>
gap> Display(B_autom);
a1 = (a1, a1), a2 = (a3, a1)(1,2), a3 = (a2, a1), a4 = (a1, a5)
(1,2), a5 = (a4, a1), a6 = (a1, a7)(1,2), a7 = (a6, a1)(1,2)
```

4 ▶ `ContractingLevel( G )` AM

Given a contracting group G with generating set N that includes the nucleus, stored in `GeneratingSetWithNucleus`(G) (see 2.5.2) computes the minimal level n, such that for every vertex v of the n-th level and all $g, h \in N$ the section $gh|_v \in N$.

In the case if it is not known whether G is contracting, it first tries to compute the nucleus. If G happens to be noncontracting, it will loop forever. One can also use `IsNoncontracting` (see 2.2.10) or `FindNucleus` (see 2.3.18) directly.

```
gap> Grigorchuk_Group := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> ContractingLevel(Grigorchuk_Group);
1
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> ContractingLevel(Basilica);
2
```

5 ▶ `ContractingTable( G )` AM

Given a contracting group G with a generating set N of size k that includes the nucleus, stored in `GeneratingSetWithNucleus`(G) (see 2.5.2) computes the $k \times k$ table, whose [i][j]-th entry contains decomposition

of $N[i]N[j]$ on the `ContractingLevel`(G) level (see 2.5.4). By construction the sections of $N[i]N[j]$ on this level belong to N. This table is used in the algorithm solving the word problem in polynomial time.

In the case if it is not known whether G is contracting it first tries to compute the nucleus. If G happens to be noncontracting, it will loop forever. One can also use `IsNoncontracting` (see 2.2.10) or `FindNucleus` (see 2.3.18) directly.

```
gap> Grigorchuk_Group := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> ContractingTable(Grigorchuk_Group);
[ [ (1, 1), (1, 1)(1,2), (a, c), (a, d), (1, b) ],
  [ (1, 1)(1,2), (1, 1), (c, a)(1,2), (d, a)(1,2), (b, 1)(1,2) ],
  [ (a, c), (a, c)(1,2), (1, 1), (1, b), (a, d) ],
  [ (a, d), (a, d)(1,2), (1, b), (1, 1), (a, c) ],
  [ (1, b), (1, b)(1,2), (a, d), (a, c), (1, 1) ] ]
```

6 ▶ `UseContraction(` G `)` O

▶ `DoNotUseContraction(` G `)` O

For a contracting automaton group G these two operations determine whether to use the algorithm of polynomial complexity solving the word problem in the group. By default it is set to *true* as soon as the nucleus of the group was computed. Sometimes when the nucleus is very big, the standard algorithm of exponential complexity is faster for short words, but this heavily depends on the group. Therefore the decision on which algorithm to use is left to the user. To use the exponential algorithm one can use the second operation `DoNotUseContraction`(G).

Note also then in order to use the polynomial time algorithm the `ContractingTable(G)` (see 2.5.5) has to be computed first, which takes some time when the nucleus is big. This attribute is computed automatically when the word problem is solved for the first time. This sometimes causes some delay.

Below we provide an example which shows that both methods can be of use.

```
gap> G := AutomatonGroup("a=(b,b)(1,2), b=(c,a), c=(a,a)");
< a, b, c >
gap> IsContracting(G);
true
gap> Size(GroupNucleus(G));
41
gap> ContractingLevel(G);
6
gap> ContractingTable(G);; time;
4719
gap> v := a*b*a*b^2*c*b*c*b^-1*a^-1*b^-1*a^-1;;
gap> w := b*c*a*b*a*b*c^-1*b^-2*a^-1*b^-1*a^-1;;
gap> UseContraction(G);;
gap> IsOne(Comm(v,w)); time;
true
110
gap> FindGroupRelations(G, 9);; time;
a^2
b^2
c^2
(b*a*b*c*a)^2
(b*(c*a)^2)^2
(b*c*b*a*(b*c)^2*a)^2
(b*(c*b*c*a)^2)^2
```

```
11578
gap> DoNotUseContraction(G);;
gap> IsOne(Comm(v,w)); time;
true
922
gap> FindGroupRelations(G, 9);; time;
a^2
b^2
c^2
(b*a*b*c*a)^2
(b*(c*a)^2)^2
(b*c*b*a*(b*c)^2*a)^2
(b*(c*b*c*a)^2)^2
23719
```

2.6 Rewriting Systems

It is possible to use basic relators in all computations performed in a self-similar group.

1 ▶ `AG_UseRewritingSystem( G[, setting] )` O

Tells whether computations in the group G should use a rewriting system. *setting* defaults to `true` if omitted. This function initially only tries to find involutions in G. See `AG_AddRelators` (2.6.2) and `AG_UpdateRewritingSystem` (2.6.3) for the ways to add more relators.

```
gap> G := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> Comm(a*b, b*a);
b^-1*a^-2*b^-1*a*b^2*a
gap> AG_UseRewritingSystem(G);
gap> Comm(a*b, b*a);
1
gap> AG_UseRewritingSystem(G, false);
gap> Comm(a*b, b*a);
b^-1*a^-2*b^-1*a*b^2*a
```

2 ▶ `AG_AddRelators( G, relators )` O

Adds relators from the list *relators* to the rewriting system used in G.

```
gap> G := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> AG_UseRewritingSystem(G);
gap> b*c;
b*c
gap> AG_AddRelators(G, [b*c*d]);
gap> b*c;
d
```

In some cases it's hard to find relations directly from the wreath recursion of a self-similar group (at least, there is no general agorithm). This function provides possibility to add relators manually. After that one can use `AG_UpdateRewritingSystem` (see 2.6.3) and `AG_UseRewritingSystem` (see 2.6.1) to use these relators in computations. In the example below we consider a finite group H, in which $a = b$, but the standard algorithm is unable to solve the word problem. There are two solutions for that. One can manually add a relator, or one can ask if the group is finite (which does not stop generally if the group is infinite).

```
gap> H := SelfSimilarGroup("a=(a*b,1)(1,2), b=(1,b*a^-1)(1,2), c=(b, a*b)");
< a, b, c >
gap> AG_AddRelators(H, [a*b^-1]);
gap> AG_UseRewritingSystem(H);
gap> Order(a*c);
4
```

3 ▶ `AG_UpdateRewritingSystem( G, maxlen )` O

Tries to find new relators of length up to *maxlen* and adds them into the rewriting system. It can also be used after introducing new relators via `AG_AddRelators` (see 2.6.2).

```
gap> G := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> AG_UseRewritingSystem(G);
gap> b*c;
b*c
gap> AG_UpdateRewritingSystem(G, 3);
gap> b*c;
d
```

4 ▶ `AG_RewritingSystemRules( G )` O

Returns the list of rules used in the rewriting system of group G.

```
gap> G := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> AG_UseRewritingSystem(G);
gap> AG_RewritingSystemRules(G);
[ [ a^2, <identity ...> ], [ b^2, <identity ...> ], [ c^2, <identity ...> ],
  [ d^2, <identity ...> ], [ A, a ], [ B, b ], [ C, c ], [ D, d ] ]
```

3 Properties and operations with group and semigroup elements

In this chapter we present the functionality applicable to elements of groups and semigroups.

3.1 Creation of tree automorphisms and homomorphisms

1 ▶ `TreeAutomorphism(` *states*, *perm* `)` O

Constructs the tree automorphism with states on the first level given by the argument *states* and acting on the first level as the permutation *perm*. The *states* must belong to the same family.

```
gap> L := AutomatonGroup("p=(p,q)(1,2), q=(p,q)");
< p, q >
gap> r := TreeAutomorphism([p, q, p, q^2],(1,2)(3,4));
(p, q, p, q^2)(1,2)(3,4)
gap> t := TreeAutomorphism([q, 1, p*q, q],(1,2));
(q, 1, p*q, q)(1,2)
gap> r*t;
(p, q^2, p*q, q^2*p*q)(3,4)
```

2 ▶ `TreeHomomorphism(` *states*, *tr* `)` O

Constructs an homomorphism with states *states* and acting on the first level with transformation *tr*. The *states* must belong to the same family.

```
gap> S := AutomatonSemigroup("a=(a,b)[1,1],b=(b,a)(1,2)");
< a, b >
gap> x := TreeHomomorphism([a,b^2,a,a*b],Transformation([3,1,2,2]));
(a, b^2, a, a*b)[3,1,2,2]
gap> y := TreeHomomorphism([a*b,b,b,b^2],Transformation([1,4,2,3]));
(a*b, b, b, b^2)[1,4,2,3]
gap> x*y;
(a*b, b^2*a*b, a*b, a*b^2)[2,1,4,4]
```

3 ▶ `Representative(` *word*, *fam* `)` O
▶ `Representative(` *word*, *a* `)` O

Given an associative word *word* constructs the tree homomorphism from the family *fam*, or to which homomorphism *a* belongs. This function is useful when one needs to make some operations with associative words. See also `Word` (3.2.12).

```
gap> L := AutomatonGroup("p=(p,q)(1,2), q=(p,q)");
< p, q >
gap> F := UnderlyingFreeGroup(L);
<free group on the generators [ p, q ]>
gap> r := Representative(F.1*F.2^2, p);
p*q^2
gap> Decompose(r);
(p*q^2, q*p^2)(1,2)
gap> H := SelfSimilarGroup("x=(x*y,x)(1,2), y=(x^-1,y)");
< x, y >
gap> F := UnderlyingFreeGroup(H);
<free group on the generators [ x, y ]>
gap> r := Representative(F.1^-1*F.2, x);
x^-1*y
gap> Decompose(r);
(x^-1*y, y^-1*x^-2)(1,2)
```

3.2 Properties and attributes of tree automorphisms and homomorphisms

1 ▶ `IsSphericallyTransitive( a )` P

Returns whether the action of a is spherically transitive (see 1.1).

2 ▶ `IsTransitiveOnLevel( a, lev )` O

Returns whether a acts transitively on level lev of the tree.

3 ▶ `IsOne( a )` O

Returns whether an automorphism a acts trivially on the tree. For contracting groups see also `UseContraction` (2.5.6) and `IsOneContr` (3.2.4).

```
gap> L := AutomatonGroup("p=(p,q)(1,2), q=(p,q)");
< p, q >
gap> IsOne(q*p^-1*q*p^-1);
true
```

4 ▶ `IsOneContr( a )` F

Returns `true` if a is trivial automorphism and `false` otherwise. Works for contracting groups only. Uses polynomial time algorithm.

5 ▶ `Order( a )` O

Computes the order of an automorphism a. In some cases it does not stop. Works always (modulo memory restrictions) for groups generated by bounded automata.

If `InfoLevel` of `InfoAutomGrp` is greater than or equal to 3 (one can set it by `SetInfoLevel( InfoAutomGrp, 3)`) and the element has infinite order, then the proof of this fact is printed.

```
gap> L := AutomatonGroup("p=(p,q)(1,2), q=(p,q)");
< p, q >
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> Order(p*q^-1);
2
gap> SetInfoLevel( InfoAutomGrp, 3);
gap> Order( u^35*v^-12*u^2*v^-3 );
```

```
#I  (u^35*v^-12*u^2*v^-3)^68719476736 has conjugate of u^2*v^-3*u^35*v^
-12 as a section at vertex [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
infinity
```

6 ▶ `OrderUsingSections( a[, max_depth] )` O

Tries to compute the order of the element a by looking at its sections of depth up to *max_depth*-th level. If *max_depth* is omitted it is assumed to be `infinity`, but then it may not stop. Also note, that if *max_depth* is not given, it searches the tree in depth first and may be trapped in some infinite ray, while specifying finite *max_depth* may produce a result by looking at a section not in that ray. For bounded automata it will always produce a result.

If `InfoLevel` of `InfoAutomGrp` is greater than or equal to 3 (one can set it by `SetInfoLevel( InfoAutomGrp, 3)`) and the element has infinite order, then the proof of this fact is printed.

```
gap> Grigorchuk_Group := AutomatonGroup("a=(1,1)(1,2),b=(a,c),c=(a,d),d=(1,b)");
< a, b, c, d >
gap> OrderUsingSections( a*b*a*c*b );
16
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> SetInfoLevel( InfoAutomGrp, 3);
gap> OrderUsingSections( u^23*v^-2*u^3*v^15, 10 );
#I  v^13*u^15 acts transitively on levels and is obtained from (u^23*v^-2*u^3*v^15)^1
    by taking sections and cyclic reductions at vertex [ 1 ]
infinity
gap> G := AutomatonGroup("a=(c,a)(1,2), b=(b,c), c=(b,a)");
< a, b, c >
gap> OrderUsingSections(b,10);
#I  b*c*a^2*b^2*c*a acts transitively on levels and is obtained from (b)^8
    by taking sections and cyclic reductions at vertex
[ 2, 2, 1, 1, 1, 1, 2, 2, 1, 1 ]
infinity
```

7 ▶ `Perm( a[, lev] )` O

Returns the permutation induced by the tree automorphism a on the level *lev* (or first level if *lev* is not given). See also `TransformationOnLevel` (3.2.10).

8 ▶ `PermOnLevel( a, k )` O

Does the same thing as `Perm` (3.2.7).

9 ▶ `PermOnLevelAsMatrix( g, lev )` F

Computes the action of the element g of a group on the *lev*-th level as a permutational matrix, in which the i-th row contains 1 at the position i^g.

```
gap> L := AutomatonGroup("p=(p,q)(1,2), q=(p,q)");
< p, q >
gap> PermOnLevel(p*q,2);
(1,4)(2,3)
gap> PermOnLevelAsMatrix(p*q, 2);
[ [ 0, 0, 0, 1 ], [ 0, 0, 1, 0 ], [ 0, 1, 0, 0 ], [ 1, 0, 0, 0 ] ]
```

10 ▶ `TransformationOnLevel( a, lev )` O
▶ `TransformationOnFirstLevel( a )` O

The first function returns the transformation induced by the tree homomorphism a on the level lev. See also `PermOnLevel` (3.2.8).

If the transformation is invertible then it returns a permutation, and `Transformation` otherwise.

`TransformationOnFirstLevel`(a) is equivalent to `TransformationOnLevel`(a, 1).

11 ▶ `TransformationOnLevelAsMatrix( g, lev )` F

Computes the action of the element g on the lev-th level as a permutational matrix, in which the i-th row contains 1 at the position i^g.

```
gap> L := AutomatonSemigroup("p=(p,q)(1,2), q=(p,q)[1,1]");
< p, q >
gap> TransformationOnLevel(p*q,2);
Transformation( [ 1, 1, 2, 2 ] )
gap> TransformationOnLevelAsMatrix(p*q,2);
[ [ 1, 0, 0, 0 ], [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 1, 0, 0 ] ]
```

12 ▶ `Word( a )` O

Returns a as an associative word (an element of the underlying free group) in the generators of the self-similar group (semigroup) to which a belongs.

```
gap> L := AutomatonGroup("p=(p,q)(1,2), q=(p,q)");
< p, q >
gap> w := Word(p*q^2*p^-1);
p*q^2*p^-1
gap> Length(w);
4
```

3.3 Operations with tree automorphisms and homomorphisms

The multiplication of tree homomorphisms is defined in the standard way

1 ▶ a `*` b

The following operations allow computation of the actions of tree homomorphisms on letters and vertices

2 ▶ *letter* `^` a
▶ *vertex* `^` a

```
gap> L := AutomatonGroup("p=(p,q)(1,2), q=(p,q)");
< p, q >
gap> 1^p;
2
gap> [1, 2, 2, 1, 2, 1]^(p*q^2);
[ 2, 1, 2, 2, 1, 2 ]
```

The operations below describe how to work with sections of tree homomorphisms.

3 ▶ `Section( a, v )` O

Returns the section of the automorphism (homomorphism) a at the vertex v. The vertex v can be a list representing the vertex, or a positive integer representing a vertex of the first level of the tree.

```
gap> L := AutomatonGroup("p=(p,q)(1,2), q=(p,q)");
< p, q >
gap> Section(p*q*p^2, [1,2,2,1,2,1]);
p^2*q^2
```

4 ▶ `Sections(` *a* `[,` *lev*`] )` O

Returns the list of sections of a at the *lev*-th level. If *lev* is omitted it is assumed to be 1.

```
gap> L := AutomatonGroup("p=(p,q)(1,2), q=(p,q)");
< p, q >
gap> Sections(p*q*p^2);
[ p*q^2*p, q*p^2*q ]
```

5 ▶ `Decompose(` *a*`[,` *k*`] )` O

Returns the decomposition of the tree homomorphism a on the k-th level of the tree, i.e. the representation of the form

$$a = (a_1, a_2, \ldots, a_{d_1 \times \cdots \times d_k})\sigma$$

where a_i are the sections of a at the k-th level, and σ is the transformation of the k-th level. If k is omitted it is assumed to be 1.

```
gap> L := AutomatonGroup("p=(p,q)(1,2), q=(p,q)");
< p, q >
gap> Decompose(p*q^2);
(p*q^2, q*p^2)(1,2)
gap> Decompose(p*q^2,3);
(p*q^2, q*p^2, p^2*q, q^2*p, p*q*p, q*p*q, p^3, q^3)(1,8,3,5)(2,7,4,6)
```

6 ▶ *a* `in` *G*

Returns whether the automorphism a belongs to the group G. In some cases it does not stop.

```
gap> L := AutomatonGroup("p=(p,q)(1,2), q=(p,q)");
< p, q >
gap> H := Group([p^2, q^2]);
< p^2, q^2 >
gap> p in H;
false
```

7 ▶ `OrbitOfVertex(` *ver*, *g*`[,` *n*`] )` O

Returns the list of vertices in the orbit of the vertex *ver* under the action of the semigroup generated by the automorphism g. If n is specified, it returns only the first n elements of the orbit. Vertices are defined either as lists with entries from $\{1, \ldots, d\}$, or as strings containing characters $1, \ldots, d$, where d is the degree of the tree.

```
gap> T := AutomatonGroup("t=(1,t)(1,2)");
< t >
gap> OrbitOfVertex([1,1,1], t);
[ [ 1, 1, 1 ], [ 2, 1, 1 ], [ 1, 2, 1 ], [ 2, 2, 1 ], [ 1, 1, 2 ],
  [ 2, 1, 2 ], [ 1, 2, 2 ], [ 2, 2, 2 ] ]
gap> OrbitOfVertex("11111111111", t, 6);
[ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ], [ 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1 ], [ 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1 ], [ 2, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1 ] ]
```

8 ▶ `PrintOrbitOfVertex( ` *ver*`, ` *g*`[, ` *n*`] )` O

Prints the orbit of the vertex *ver* under the action of the semigroup generated by g. Each vertex is printed as a string containing characters $1, \ldots, d$, where d is the degree of the tree. In case of binary tree the symbols " " and "x" are used to represent 1 and 2. If n is specified only the first n elements of the orbit are printed. Vertices are defined either as lists with entries from $\{1, \ldots, d\}$, or as strings. See also `OrbitOfVertex` (3.3.7).

```
gap> L := AutomatonGroup("p=(p,q)(1,2), q=(p,q)");
< p, q >
gap> PrintOrbitOfVertex("22222222222222222222222222222", p*q^-2, 6);
xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 x x x x x x x x x x x x x x
x  xx  xx  xx  xx  xx  xx  xx
   x   x   x   x   x   x   x
xxx    xxxx    xxxx    xxxx
 x     x x     x x     x x
gap> H := AutomatonGroup("t=(s,1,1)(1,2,3), s=(t,s,t)(1,2)");
< t, s >
gap> PrintOrbitOfVertex([1,2,1], s^2);
121
132
123
131
122
133
```

9 ▶ `PermActionOnLevel( ` *perm*`, ` *big_lev*`, ` *sm_lev*`, ` *deg* `)` F

Given a permutation *perm* on the *big_lev*-th level of the tree of degree *deg* returns the permutation induced by *perm* on a smaller level *sm_lev*.

```
gap> PermActionOnLevel((1,4,2,3), 2, 1, 2);
(1,2)
gap> PermActionOnLevel((1,13,5,9,3,15,7,11)(2,14,6,10,4,16,8,12), 4, 2, 2);
(1,4,2,3)
```

3.4 Elements of groups and semigroups defined by wreath recursion

1 ▶ `IsFiniteState( ` *a* `)` P

Returns `true` if a has finitely many different sections. It will never stop if the free reduction of words is not sufficient to establish the finite-state property or if a is not finite-state (has infinitely many different sections).

See also `AllSections` (3.4.2) for the list of all sections and `MealyAutomaton` (4.1.1), which allows to construct a Mealy automaton whose states are the sections of a and which encodes its action on the tree.

```
gap> D := SelfSimilarGroup("x=(1,y)(1,2), y=(z^-1,1)(1,2), z=(1,x*y)");
< x, y, z >
gap> IsFiniteState(x*y^-1);
true
```

2 ▶ `AllSections( ` *a* `)` A

Returns the list of all sections of a if there are finitely many of them and this fact can be established using free reduction of words in sections. Otherwise will never stop. Note, that in the case when a is an element of a self-similar (semi)group defined by wreath recurion it does not check whether all elements of the list

are actually different automorphisms (homomorphisms) of the tree. If a is a element of of a (semi)group generated by finite automaton, it will always return the list of all distinct sections of a.

```
gap> D := SelfSimilarGroup("x=(1,y)(1,2), y=(z^-1,1)(1,2), z=(1,x*y)");
< x, y, z >
gap> AllSections(x*y^-1);
[ x*y^-1, z, 1, x*y, y*z^-1, z^-1*y^-1*x^-1, y^-1*x^-1*z*y^-1, z*y^-1*x*y*z,
  y*z^-1*x*y, z^-1*y^-1*x^-1*y*z^-1, x*y*z, y, z^-1, y^-1*x^-1, z*y^-1 ]
```

See also operation `MealyAutomaton` (4.1.1), which allows to construct a Mealy automaton whose states are the sections of given tree homomorphism and which encodes its action on the tree.

3.5 Elements of contracting groups

1 ▶ `AutomPortrait( a )` F
▶ `AutomPortraitBoundary( a )` F
▶ `AutomPortraitDepth( a )` F

Constructs the portrait of an element a of a contracting group G. The portrait of a is defined recursively as follows. For g in the nucleus of G the portrait is just $[g]$. For any other element $g = (g_1, g_2, \ldots, g_d)\sigma$ the portrait of g is $[\sigma, \texttt{AutomPortrait}(g_1), \ldots, \texttt{AutomPortrait}(g_d)]$, where d is the degree of the tree. This structure describes a finite tree whose inner vertices are labelled by permutations from S_d and the leaves are labelled by elements from the nucleus. The contraction in G guarantees that the portrait of any element is finite.

The portraits may be considered as "normal forms" of the elements of G, since different elements have different portraits.

One also can be interested only in the boundary of a portrait, which consists of all leaves of the portrait. This boundary can be described by an ordered set of pairs $[level_i, g_i]$, $i = 1, \ldots, r$ representing the leaves of the tree ordered from left to right (where $level_i$ and g_i are the level and the label of the i-th leaf correspondingly, r is the number of leaves). The operation `AutomPortraitBoundary`(a) computes this boundary.

`AutomPortraitDepth`(a) returns the depth of the portrait, i.e. the minimal level such that all sections of a at this level belong to the nucleus of G.

```
gap> Basilica := AutomatonGroup("u=(v,1)(1,2), v=(u,1)");
< u, v >
gap> AutomPortrait(u^3*v^-2*u);
[ (), [ (), [ (), [ v ], [ v ] ], [ 1 ] ],
  [ (), [ (), [ v ], [ u^-1*v ] ], [ v^-1 ] ] ]
gap> AutomPortrait(u^3*v^-2*u^3);
[ (), [ (), [ (1,2), [ (), [ (), [ v ], [ v ] ], [ 1 ] ], [ v ] ], [ 1 ] ],
  [ (), [ (1,2), [ (), [ (), [ v ], [ v ] ], [ 1 ] ], [ u^-1*v ] ], [ v^-1 ]
     ] ]
gap> AutomPortraitBoundary(u^3*v^-2*u^3);
[ [ 5, v ], [ 5, v ], [ 4, 1 ], [ 3, v ], [ 2, 1 ], [ 5, v ], [ 5, v ],
  [ 4, 1 ], [ 3, u^-1*v ], [ 2, v^-1 ] ]
gap> AutomPortraitDepth(u^3*v^-2*u^3);
5
```

4 Noninitial automata

In this chapter we present the functionality applicable to noninitial automata.

4.1 Definition

1 ▶ `MealyAutomaton(` *table*`[,` *names*`[,` *alphabet*`]] )` O
▶ `MealyAutomaton(` *string* `)` O
▶ `MealyAutomaton(` *autom* `)` O
▶ `MealyAutomaton(` *tree_hom_list* `)` O
▶ `MealyAutomaton(` *list*`,` *name_func* `)` O
▶ `MealyAutomaton(` *list*`,` *true* `)` O

Creates the Mealy automaton (see 1.1) defined by the argument *table*, *string* or *autom*. Format of the argument *table* is the following: it is a list of states, where each state is a list of positive integers which represent transition function at the given state and a permutation or transformation which represent the output function at this state. Format of the string *string* is the same as in `AutomatonGroup` (see 2.1.1). The third form of this operation takes a tree homomorphism *autom* as its argument. It returns noninitial automaton constructed from the sections of *autom*, whose first state corresponds to *autom* itself. The fourth form creates a noninitial automaton constructed of the states of all tree homomorphisms from the *tree_hom_list*.

```
gap> A := MealyAutomaton([[1,2,(1,2)],[3,1,()],[3,3,(1,2)]], ["a","b","c"]);
<automaton>
gap> Display(A);
a = (a, b)(1,2), b = (c, a), c = (c, c)(1,2)
gap> B:=MealyAutomaton([[1,2,Transformation([1,1])],[3,1,()],[3,3,(1,2)]],["a","b","c"]);
<automaton>
gap> Display(B);
a = (a, b)[ 1, 1 ], b = (c, a), c = (c, c)[ 2, 1 ]
gap> D := MealyAutomaton("a=(a,b)(1,2), b=(b,a)");
<automaton>
gap> Display(D);
a = (a, b)(1,2), b = (b, a)
gap> Basilica := AutomatonGroup( "u=(v,1)(1,2), v=(u,1)" );
< u, v >
gap> M := MealyAutomaton(u*v*u^-3);
<automaton>
gap> Display(M);
a1 = (a2, a5), a2 = (a3, a4), a3 = (a4, a2)(1,2), a4 = (a4, a4), a5 = (a6, a3)
(1,2), a6 = (a7, a4), a7 = (a6, a4)(1,2)
```

If *list* consists of tree homomorphisms, it creates a noninitial automaton constructed of their states. If *name_func* is a function then it is used to name the states of the newly constructed automaton. If it is *true* then states of automata from the *list* are used. If it *false* then new states are named a_1, a_2, etc.

```
gap> G := AutomatonGroup("a=(b,a),b=(b,a)(1,2)");
< a, b >
gap> MealyAutomaton([a*b]);; Display(last);
a1 = (a2, a4)(1,2), a2 = (a3, a1), a3 = (a3, a1)(1,2), a4 = (a2, a4)
gap> MealyAutomaton([a*b], true);; Display(last);
<a*b> = (<b^2>, <a^2>)(1,2), <b^2> = (<b*a>, <a*b>), <b*a> = (<b*a>, <a*b>)(1,2), <a^2> = (<b^2>, <a
gap> MealyAutomaton([a*b], String);; Display(last);
a*b = (b^2, a^2)(1,2), b^2 = (b*a, a*b), b*a = (b*a, a*b)(1,2), a^2 = (b^2, a^2)
```

2 ▶ `IsMealyAutomaton( A )` C

A category of non-initial finite Mealy automata with the same input and output alphabet.

3 ▶ `NumberOfStates( A )` A

Returns the number of states of the automaton A.

4 ▶ `SizeOfAlphabet( A )` A

Returns the number of letters in the alphabet the automaton A acts on.

5 ▶ `AutomatonList( A )` A

Returns the list of A acceptable by `MealyAutomaton` (see 4.1.1)

4.2 Tools

1 ▶ `IsTrivial( A )` P

Computes whether the automaton A is equivalent to the trivial automaton.

```
gap> A := MealyAutomaton("a=(c,c), b=(a,b), c=(b,a)");
<automaton>
gap> IsTrivial(A);
true
```

2 ▶ `IsInvertible( A )` P

Is `true` if A is invertible and `false` otherwise.

3 ▶ `MinimizationOfAutomaton( A )` F

Returns the automaton obtained from automaton A by minimization. The implementation of this function was significantly optimized by Andrey Russev starting from Version 1.3.

```
gap> B := MealyAutomaton("a=(1,a)(1,2), b=(1,a)(1,2), c=(a,b), d=(a,b)");
<automaton>
gap> C := MinimizationOfAutomaton(B);
<automaton>
gap> Display(C);
a = (1, a)(1,2), c = (a, a), 1 = (1, 1)
```

4 ▶ `MinimizationOfAutomatonTrack( A )` F

Returns the list `[A_new, new_via_old, old_via_new]`, where `A_new` is an automaton obtained from automaton A by minimization, `new_via_old` describes how new states are expressed in terms of the old ones, and `old_via_new` describes how old states are expressed in terms of the new ones. The implementation of this function was significantly optimized by Andrey Russev starting from Version 1.3.

```
gap> B := MealyAutomaton("a=(1,a)(1,2), b=(1,a)(1,2), c=(a,b), d=(a,b)");
<automaton>
gap> B_min := MinimizationOfAutomatonTrack(B);
[ <automaton>, [ 1, 3, 5 ], [ 1, 1, 2, 2, 3 ] ]
gap> Display(B_min[1]);
a = (1, a)(1,2), c = (a, a), 1 = (1, 1)
```

5 ► `IsOfPolynomialGrowth( A )` P

Determines whether the automaton A has polynomial growth in terms of Sidki [Sid00].

See also `IsBounded` (4.2.6) and `PolynomialDegreeOfGrowth` (4.2.7).

```
gap> B := MealyAutomaton("a=(b,1)(1,2), b=(a,1)");
<automaton>
gap> IsOfPolynomialGrowth(B);
true
gap> D := MealyAutomaton("a=(a,b)(1,2), b=(b,a)");
<automaton>
gap> IsOfPolynomialGrowth(D);
false
```

6 ► `IsBounded( A )` P

Determines whether the automaton A is bounded in terms of Sidki [Sid00].

See also `IsOfPolynomialGrowth` (4.2.5) and `PolynomialDegreeOfGrowth` (4.2.7).

```
gap> B := MealyAutomaton("a=(b,1)(1,2), b=(a,1)");
<automaton>
gap> IsBounded(B);
true
gap> C := MealyAutomaton("a=(a,b)(1,2), b=(b,c), c=(c,1)(1,2)");
<automaton>
gap> IsBounded(C);
false
```

7 ► `PolynomialDegreeOfGrowth( A )` A

For an automaton A of polynomial growth in terms of Sidki [Sid00] determines its degree of polynomial growth. This degree is 0 if and only if automaton is bounded. If the growth of automaton is exponential returns `fail`.

See also `IsOfPolynomialGrowth` (4.2.5) and `IsBounded` (4.2.6).

```
gap> B := MealyAutomaton("a=(b,1)(1,2), b=(a,1)");
<automaton>
gap> PolynomialDegreeOfGrowth(B);
0
gap> C := MealyAutomaton("a=(a,b)(1,2), b=(b,c), c=(c,1)(1,2)");
<automaton>
gap> PolynomialDegreeOfGrowth(C);
2
```

8 ► `AdjacencyMatrix( A )` A

Returns the adjacency matrix of a Mealy automaton A, in which the ij-th entry contains the number of arrows in the Moore diagram of A from state i to state j.

```
gap> A:=MealyAutomaton("a=(a,a,b)(1,2,3),b=(a,c,b)(1,2),c=(a,a,a)");
<automaton>
gap> AdjacencyMatrix(A);
[ [ 2, 1, 0 ], [ 1, 1, 1 ], [ 3, 0, 0 ] ]
```

9 ▶ `IsAcyclic( A )` P

Computes whether or not an automaton A is acyclic in the sense of Sidki [Sid00]. I.e. returns `true` if the Moore diagram of A does not contain cycles with two or more states and `false` otherwise.

```
gap> A := MealyAutomaton("a=(a,a,b)(1,2,3),b=(c,c,b)(1,2),c=(d,c,1),d=(d,1,d)");
<automaton>
gap> IsAcyclic(A);
true
gap> A := MealyAutomaton("a=(a,a,b)(1,2,3),b=(c,c,d)(1,2),c=(d,c,1),d=(b,1,d)");
<automaton>
gap> IsAcyclic(A);
false
```

10 ▶ `DualAutomaton( A )` O

Returns the automaton dual of A.

```
gap> A := MealyAutomaton("a=(b,a)(1,2), b=(b,a)");
<automaton>
gap> D := DualAutomaton(A);
<automaton>
gap> Display(D);
d1 = (d2, d1)[ 2, 2 ], d2 = (d1, d2)[ 1, 1 ]
```

11 ▶ `InverseAutomaton( A )` O

Returns the automaton inverse to A if A is invertible.

```
gap> A := MealyAutomaton("a=(b,a)(1,2), b=(b,a)");
<automaton>
gap> B := InverseAutomaton(A);
<automaton>
gap> Display(B);
a1 = (a1, a2)(1,2), a2 = (a2, a1)
```

12 ▶ `IsBireversible( A )` P

Computes whether or not the automaton A is bireversible, i.e. A, the dual of A and the dual of the inverse of A are invertible. The example below shows that the Bellaterra automaton is bireversible.

```
gap> Bellaterra := MealyAutomaton("a=(c,c)(1,2), b=(a,b), c=(b,a)");
<automaton>
gap> IsBireversible(Bellaterra);
true
```

13 ▶ `IsReversible( A )` P

Computes whether or not the automaton A is reversible, i.e. the dual of A is invertible.

14 ▶ `IsIRAutomaton( A )` P

Computes whether or not the automaton A is an IR-automaton, i.e. A and its dual are invertible. The example below shows that the automaton generating lamplighter group is an IR-automaton.

```
gap> L := MealyAutomaton("a=(b,a)(1,2), b=(a,b)");
<automaton>
gap> IsIRAutomaton(L);
true
```

The next three commands deal with the, so-called, MD-reduction procedure developed in [AKL+12]. Particularly, according to [Kli13], a 2-letter or 2-state invertible reversible automaton generates a finite group if and only if the MD-reduction procedure terminates with the trivial automaton. In this case an automaton is called MD-trivial.

15 ▶ `MDReduction( A )` O

Performs the process of MD-reduction of automaton A (alternating applications of minimization and dualization procedures) until a pair of minimal automata dual to each other is reached. Returns this pair. The main point of this procedure is in the fact that the (semi)group generated by the original automaton is finite if and only each of the (semi)groups generated by the output automata is finite.

```
gap> A:=MealyAutomaton("a=(d,d,d,d)(1,2)(3,4),b=(b,b,b,b)(1,4)(2,3),\\
>                      c=(a,c,a,c),          d=(c,a,c,a)");
<automaton>
gap> NumberOfStates(MinimizationOfAutomaton(A));
4
gap> MDR:=MDReduction(A);
[ <automaton>, <automaton> ]
gap> Display(MDR[1]);
d1 = (d2, d2, d1, d1)(1,4,3), d2 = (d1, d1, d2, d2)(1,4)
gap> Display(MDR[2]);
d1 = (d4, d4)(1,2), d2 = (d2, d2)(1,2), d3 = (d1, d3), d4 = (d3, d1)
```

16 ▶ `IsMDTrivial( A )` P

Returns `true` if A is MD-trivial (i.e. if MD-reduction proedure returns the trivial automaton) and `false` otherwise.

17 ▶ `IsMDReduced( A )` P

Returns `true` if A is MD-reduced and `false` otherwise.

18 ▶ `DisjointUnion( A, B )` O

Constructs the disjoint union of automata A and B

```
gap> A := MealyAutomaton("a=(a,b)(1,2), b=(a,b)");
<automaton>
gap> B := MealyAutomaton("c=(d,c), d=(c,e)(1,2), e=(e,d)");
<automaton>
gap> Display(DisjointUnion(A, B));
a1 = (a1, a2)(1,2), a2 = (a1, a2), a3 = (a4, a3), a4 = (a3, a5)
(1,2), a5 = (a5, a4)
```

19 ▶ $A * B$

Constructs the product of 2 noninitial automata A and B.

```
gap> A := MealyAutomaton("a=(a,b)(1,2), b=(a,b)");
<automaton>
gap> B := MealyAutomaton("c=(d,c), d=(c,e)(1,2), e=(e,d)");
<automaton>
gap> Print(A*B);
a1 = (a1, a5)(1,2), a2 = (a3, a4), a3 = (a2, a6)
(1,2), a4 = (a2, a4), a5 = (a1, a6)(1,2), a6 = (a3, a5)
```

20 ▶ `SubautomatonWithStates(` *A*, *states* `)` O

Returns the minimal subautomaton of the automaton *A* containing states *states*.

```
gap> A := MealyAutomaton("a=(e,d)(1,2),b=(c,c),c=(b,c)(1,2),d=(a,e)(1,2),e=(e,d)");
<automaton>
gap> Display(SubautomatonWithStates(A, [1, 4]));
a = (e, d)(1,2), d = (a, e)(1,2), e = (e, d)
```

21 ▶ `AutomatonNucleus(` *A* `)` O

Returns the nucleus of the automaton *A*, i.e. the minimal subautomaton containing all cycles in *A*.

```
gap> A := MealyAutomaton("a=(b,c)(1,2),b=(d,d),c=(d,b)(1,2),d=(d,b)(1,2),e=(a,d)");
<automaton>
gap> Display(AutomatonNucleus(A));
b = (d, d), d = (d, b)(1,2)
```

22 ▶ `AreEquivalentAutomata(` *A*, *B* `)` O

Returns `true` if for every state `s` of the automaton *A* there is a state of the automaton *B* equivalent to `s` and vice versa.

```
gap> A := MealyAutomaton("a=(b,a)(1,2), b=(a,c), c=(b,c)(1,2)");
<automaton>
gap> B := MealyAutomaton("b=(a,c), c=(b,c)(1,2), a=(b,a)(1,2), d=(b,c)(1,2)");
<automaton>
gap> AreEquivalentAutomata(A, B);
true
```

5 Miscellaneous

In this chapter we present the functionality that does not quite fit in other chapters and the list of predefined groups and semigroups.

5.1 Converters to and from FR package

1 ▶ `FR2AutomGrp` O

This operation is designed to convert data structures defined in FR package written by Laurent Bartholdi to corresponding structures in AutomGrp package. Currently it is implemented for functionally recursive groups, semigroups, and their sub(semi)groups and elements.

```
gap> ZZ := FRGroup("t=<,t>[2,1]");
<state-closed group over [ 1 .. 2 ] with 1 generator>
gap> AZZ := FR2AutomGrp(ZZ);
< t >
gap> Display(AZZ);
< t = (1, t)(1,2) >

gap> i4 := FRMonoid("s=(1,2)","f=<s,f>[1,1]");
<state-closed monoid over [ 1 .. 2 ] with 2 generators>
gap> Ai4 := FR2AutomGrp(i4);
< 1, s, f >
gap> Display(Ai4);
< 1 = (1, 1),
  s = (1, 1)(1,2),
  f = (s, f)[1,1] >

gap> S := FRGroup("a=<a*b^-2,b^3>(1,2)","b=<b^-1*a,1>");
<state-closed group over [ 1 .. 2 ] with 2 generators>
gap> AS := FR2AutomGrp(S);
< a, b >
gap> Display(AS);
< a = (a*b^-2, b^3)(1,2),
  b = (b^-1*a, 1) >
gap> AssignGeneratorVariables(S);
#I  Global variable ‘a’ is already defined and will be overwritten
#I  Global variable ‘b’ is already defined and will be overwritten
#I  Assigned the global variables [ "a", "b" ]
gap> x := a^3*b*a^-2;
<2|a^3*b*a^-2>
gap> DecompositionOfFRElement(x);
[ [ <2|a*b^-2>, <2|b^3*a^2*b^-1*a^-1> ], [ 2, 1 ] ]
gap> y := FR2AutomGrp(x);
```

```
a^3*b*a^-2
gap> Decompose(y);
(a*b^-2, b^3*a^2*b^-1*a^-1)(1,2)
```

2 ▶ `AutomGrp2FR` O

This operation is designed to convert data structures defined in AutomGrp to corresponding structures in AutomGrp package written by Laurent Bartholdi. Currently it is implemented for automaton and self-similari (or, functionally recursive in L.Bartholdi's terminology) groups, semigroups, their sub(semi)groups and elements.

```
gap> G:=AutomatonGroup("a=(b,a)(1,2),b=(a,b)");
< a, b >
gap> FG := AutomGrp2FR(G);
<state-closed group over [ 1 .. 2 ] with 2 generators>
gap> DecompositionOfFRElement(FG.1);
[ [ <2|b>, <2|a> ], [ 2, 1 ] ]
gap> DecompositionOfFRElement(FG.2);
[ [ <2|a>, <2|b> ], [ 1, 2 ] ]

gap> G := SelfSimilarGroup("a=(a*b^-2,b*a)(1,2),b=(b^-1,a*b*a)");
< a, b >
gap> F := AutomGrp2FR(G);
<state-closed group over [ 1 .. 2 ] with 1 generator>
gap> DecompositionOfFRElement(F.1);
[ [ <2|a*b^-2>, <2|b*a> ], [ 2, 1 ] ]

gap> G := AutomatonGroup("a=(b,a)(1,2),b=(a,b),c=(c,a)(1,2)");
< a, b, c >
gap> H := Group([a*b,b*c^-2,a]);
< a*b, b*c^-2, a >
gap> FH := AutomGrp2FR(H);
<recursive group over [ 1 .. 2 ] with 3 generators>
gap> DecompositionOfFRElement(FH.1);
[ [ <2|b^2>, <2|a^2> ], [ 2, 1 ] ]

gap> G := SelfSimilarSemigroup("a=(a*b^2,b*a)[1,1],b=(b,a*b*a)(1,2)");
< a, b >
gap> S := AutomGrp2FR(G);
<state-closed semigroup over [ 1 .. 2 ] with 2 generators>
gap> DecompositionOfFRElement(S.1);
[ [ <2|a*b^2>, <2|b*a> ], [ 1, 1 ] ]

gap> G := AutomatonGroup("a=(b,a)(1,2),b=(a,b),c=(c,a)(1,2)");
< a, b, c >
gap> Decompose(a*b^-2);
(b^-1, a^-1)(1,2)
gap> x := AutomGrp2FR(a*b^-2);
<2|a*b^-2>
gap> DecompositionOfFRElement(x);
[ [ <2|b^-1>, <2|a^-1> ], [ 2, 1 ] ]
```

5.2 Trees

1 ▶ `NumberOfVertex( ver, deg )` F

One can naturally enumerate all the vertices of the n-th level of the tree by the numbers $1, \ldots, deg^n$. This function returns the number that corresponds to the vertex *ver* of the *deg*-ary tree. The vertex can be defined either as a list or as a string.

```
gap> NumberOfVertex([1,2,1,2], 2);
6
gap> NumberOfVertex("333", 3);
27
```

2 ▶ `VertexNumber( num, lev, deg )` F

One can naturally enumerate all the vertices of the *lev*-th level of the *deg*-ary tree by the numbers $1, \ldots, deg^n$. This function returns the vertex of this level that has number *num*.

```
gap> VertexNumber(1, 3, 2);
[ 1, 1, 1 ]
gap> VertexNumber(4, 4, 3);
[ 1, 1, 2, 1 ]
```

5.3 Some predefined groups

Several groups are predefined as fields in the global variable `AG_Groups`. Here is how to access, for example, Grigorchuk group

```
gap> G:=AG_Groups.GrigorchukGroup;
< a, b, c, d >
```

To perform operations with elements of `G` one can use `AssignGeneratorVariables` function.

```
gap> AssignGeneratorVariables(G);
#I  Global variable ‘a’ is already defined and will be overwritten
#I  Global variable ‘b’ is already defined and will be overwritten
#I  Global variable ‘c’ is already defined and will be overwritten
#I  Global variable ‘d’ is already defined and will be overwritten
#I  Assigned the global variables [ a, b, c, d ]
gap> Decompose(a*b);
(c, a)(1,2)
```

Below is a list of all predefined groups with short description and references.

1 ▶ `GrigorchukGroup` V

is the first Grigorchuk group, an infinite 2-group of intermediate growth constructed in [Gri80] (see [Gri05] for a survey on this group). It is defined as the group generated by the automaton

$$a = (1,1)(1,2), \quad b = (a,c), \quad c = (a,d), \quad d = (1,b).$$

The group is stored in the global variable `AG_Groups.GrigorchukGroup`

2 ▶ `UniversalGrigorchukGroup` V

is the universal group for the family of groups G_ω (see [Gri84]). It is defined as a group acting on the 6-ary tree, generated by the automaton

$$a = (1,1,1,1,1,1)(1,2), \quad b = (a,a,1,b,b,b), \quad c = (a,1,a,c,c,c), \quad d = (1,a,a,d,d,d).$$

The group is stored in the global variable `AG_Groups.UniversalGrigorchukGroup`

3 ▶ `Basilica` V

is the Basilica group. It was first studied in [GZ02a] and [GZ02b]. Later it became the first example of amenable, but not subexponentially amenable group (see [BV05]). It is the iterated monodromy group of the map $f(z) = z^2 - 1$. It is generated by the automaton

$$u = (v, 1)(1, 2), \quad v = (u, 1).$$

The group is stored in the global variable `AG_Groups.Basilica`

4 ▶ `Lamplighter` V

is the lamplighter group. This group was the key ingredient in the counterexample to the strong Atiyah conjecture (see [GLSZ00]). It is generated by the automaton

$$a = (a, b)(1, 2), \quad b = (a, b).$$

The group is stored in the global variable `AG_Groups.Lamplighter`

5 ▶ `AddingMachine` V

is the free abelian group of rank 1 (see [GNS00]) generated by the automaton

$$a = (1, a)(1, 2).$$

The group is stored in the global variable `AG_Groups.AddingMachine`

6 ▶ `InfiniteDihedral` V

is the infinite dihedral group (see [GNS00]) generated by the automaton

$$a = (a, a)(1, 2), \quad b = (b, a).$$

The group is stored in the global variable `AG_Groups.InfiniteDihedral`

7 ▶ `AleshinGroup` V

is a group generated by the Aleshin automaton (see [Ale83]) defined by the following wreath recursion:

$$a = (b, c)(1, 2), \quad b = (c, b)(1, 2), \quad c = (a, a).$$

It is isomorphic to the free group of rank 3 as was proved by M.Vorobets and Y.Vorobets (see [VV07]). The group is stored in the global variable `AG_Groups.AleshinGroup`

8 ▶ `Bellaterra` V

is a group generated by the Aleshin automaton (see [Ale83]) defined by the following wreath recursion:

$$a = (c, c)(1, 2), \quad b = (a, b), \quad c = (b, a).$$

It is isomorphic to the free product of 3 cyclic groups of order 2 (see [BGK+09]) The group is stored in the global variable `AG_Groups.Bellaterra`

9 ▶ `SushchanskyGroup` V

is the self-similar closure of the faithful level-transitive action of the Sushchansky group on the ternary tree. The original groups constructed in [Sus79] are infinite p-groups of intermediate growth acting on the p-ary

tree. In [BS07] the action of these groups on the tree was simplified, so that, in particular, the self-similar closure of one of the 3-groups is generated by the automaton

$$A = (1,1,1)(1,2,3), \quad A^2 = (1,1,1)(1,3,2), \quad B = (r_1, q_1, A),$$

$$\begin{aligned} r_1 = (r_2, A, 1), \quad r_2 = (r_3, 1, 1), \quad r_3 = (r_4, 1, 1),\\ r_4 = (r_5, A, 1), \quad r_5 = (r_6, A^2, 1), \quad r_6 = (r_7, A, 1),\\ r_7 = (r_8, A, 1), \quad r_8 = (r_9, A, 1), \quad r_9 = (r_1, A^2, 1),\\ q_1 = (q_2, 1, 1), \quad q_2 = (q_3, A, 1), \quad q_3 = (q_1, A, 1). \end{aligned}$$

The group $\langle A, B\rangle$ is isomorphic to the original Sushchansky 3-group. The group is stored in the global variable `AG_Groups.SushchanskyGroup`

10 ▶ `Hanoi3` V
▶ `Hanoi4` V

Groups related to the Hanoi towers game on 3 and 4 pegs correspondingly (see [GŠ06] and [GŠ08]). For 3 pegs `Hanoi3` is generated by the automaton

$$a_{23} = (a_{23}, 1, 1)(2,3), \quad a_{13} = (1, a_{13}, 1)(1,3), \quad a_{12} = (1, 1, a_{12})(1,2),$$

while the automaton generating `Hanoi4` is

$$a_{12} = (1,1,a_{12},a_{12})(1,2), \quad a_{13} = (1,a_{13},1,a_{13})(1,3), \quad a_{14} = (1,a_{14},a_{14},1)(1,4),$$

$$a_{23} = (a_{23},1,1,a_{23})(2,3), \quad a_{24} = (a_{24},1,a_{24},1)(2,4), \quad a_{34} = (a_{34},a_{34},1,1)(3,4).$$

The groups are stored in the global variables `AG_Groups.Hanoi3` and `AG_Groups.Hanoi4`

11 ▶ `GuptaSidki3Group` V

is the Gupta-Sidki infinite 3-group constructed in [GS83] and generated by the automaton

$$a = (1,1,1)(1,2,3), \quad b = (a, a^{-1}, b).$$

The group is stored in the global variable `AG_Groups.GuptaSidki3Group`

12 ▶ `GuptaFabrikowskiGroup` V

is the Gupta-Fabrykowski group of intermediate growth constructed in [FG85] and generated by the automaton

$$a = (1,1,1)(1,2,3), \quad b = (a,1,b).$$

The group is stored in the global variable `AG_Groups.GuptaFabrikowskiGroup`

13 ▶ `BartholdiGrigorchukGroup` V

is the Bartholdi-Grigorchuk group studied in [BG02] and generated by the automaton

$$a = (1,1,1)(1,2,3), \quad b = (a,a,b).$$

The group is stored in the global variable `AG_Groups.BartholdiGrigorchukGroup`

14 ▶ `GrigorchukErschlerGroup` V

is the group of subexponential growth studied by Erschler in [Ers04]. It grows faster than $\exp(n^\alpha)$ for any $\alpha < 1$. It belongs to the class of groups constructed by Grigorchuk in [Gri84] and corresponds to the sequence 01010101 ... . It is generated by the automaton

$$a = (1,1)(1,2), \quad b = (a,b), \quad c = (a,d), \quad d = (1,c).$$

The group is stored in the global variable `AG_Groups.GrigorchukErschlerGroup`

15 ▶ `BartholdiNonunifExponGroup` V

is the group of nonuniformly exponential growth constructed by Bartholdi in [Bar03]. Similar examples were constructed earlier in [Wil04] by Wilson. It is generated by the automaton

$$x = (1,1,1,1,1,1,1)(1,5)(3,7), \quad y = (1,1,1,1,1,1,1)(2,3)(6,7), \quad z = (1,1,1,1,1,1,1)(4,6)(5,7),$$

$$x_1 = (x_1, x, 1, 1, 1, 1, 1), \quad y_1 = (y_1, y, 1, 1, 1, 1, 1), \quad z_1 = (z_1, z, 1, 1, 1, 1, 1).$$

The group is stored in the global variable `AG_Groups.BartholdiNonunifExponGroup`

16 ▶ `IMG_z2plusI` V

The iterated monodromy group of the map $f(z) = z^2 + i$. It has intermediate growth (see [BP06]) and was studied in [GSŠ07].

$$a = (1,1)(1,2), \quad b = (a,c), c = (b,1).$$

The group is stored in the global variable `AG_Groups.IMG_z2plusI`

17 ▶ `Airplane` V
▶ `Rabbit` V

These are iterated monodromy groups of certain quadratic polynomials studied in [BN06]. It was proved there that they are not isomorphic. The automata generating them are the following

$$a = (b,1)(1,2), \quad b = (c,1), \quad c = (a,1);$$

$$a = (b,1)(1,2), \quad b = (1,c), \quad c = (a,1).$$

The groups are stored in the global variables `AG_Groups.Airplane` and `AG_Groups.Rabbit`

18 ▶ `TwoStateSemigroupOfIntermediateGrowth` V

is the semigroup of intermediate growth studied in [BRS06]. It is generated by the automaton

$$f_0 = (f_0, f_0)(1,2), \quad f_1 = (f_1, f_0)[2,2]\cdot$$

The group is stored in the global variable `AG_Groups.TwoStateSemigroupOfIntermediateGrowth`

19 ▶ `UniversalD_omega` V

is the group constructed in [Nek07] as the universal group which covers an uncountable family of groups parameterized by infinite binary sequences. It is contracting with nucleus consisting of 35 elements. Its generating automaton is as follows (it acts on the 4-ary tree):

$$a = (1,2)(3,4), \quad b = (a,c,a,c), \quad c = (b,1,1,b).$$

The group is stored in the global variable `AG_Groups.UniversalD_omega`

Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., "`PermutationCharacter`" comes before "permutation group".

Bibliography

[AKL+12] Ali Akhavi, Ines Klimann, Sylvain Lombardy, Jean Mairesse, and Matthieu Picantin. On the finiteness problem for automaton (semi)groups. *Internat. J. Algebra Comput.*, 22(6):1250052, 26, 2012.

[Ale83] S. V. Aleshin. A free group of finite automata. *Vestnik Moskov. Univ. Ser. I Mat. Mekh.*, (4):12–14, 1983.

[Bar03] Laurent Bartholdi. A Wilson group of non-uniformly exponential growth. *C. R. Math. Acad. Sci. Paris*, 336(7):549–554, 2003.

[BG02] Laurent Bartholdi and Rostislav I. Grigorchuk. On parabolic subgroups and Hecke algebras of some fractal groups. *Serdica Math. J.*, 28(1):47–90, 2002.

[BGK+08] I. Bondarenko, R. Grigorchuk, R. Kravchenko, Y. Muntyan, V. Nekrashevych, D. Savchuk, and Z. Šunić. Classification of groups generated by 3-state automata over 2-letter alphabet. *Algebra Discrete Math.*, (1):1–163, 2008.

[BGK+09] I. Bondarenko, R. Grigorchuk, R. Kravchenko, Y. Muntyan, V. Nekrashevych, D. Savchuk, and Z. Šunić. Groups generated by 3-state automata over a 2-letter alphabet. II. *J. Math. Sci. (N. Y.)*, 156(1):187–208, 2009. Functional analysis.

[BKN10] Laurent Bartholdi, Vadim Kaimanovich, and Volodymyr Nekrashevych. On amenability of automata groups. *Duke Mathematical Journal*, 154(3):575–598, 2010.

[BN06] Laurent I. Bartholdi and Volodymyr V. Nekrashevych. Thurston equivalence of topological polynomials. *Acta Math.*, 197(1):1–51, 2006.

[BP06] Kai-Uwe Bux and Rodrigo Pérez. On the growth of iterated monodromy groups. In *Topological and asymptotic aspects of group theory*, volume 394 of *Contemp. Math.*, pages 61–76. Amer. Math. Soc., Providence, RI, 2006. (available at http://www.arxiv.org/abs/math.GR/0405456).

[BRS06] L. Bartholdi, I. I. Reznykov, and V. I. Sushchansky. The smallest Mealy automaton of intermediate growth. *J. Algebra*, 295(2):387–414, 2006.

[BS07] Ievgen V. Bondarenko and Dmytro M. Savchuk. On Sushchansky p-groups. *Algebra Discrete Math.*, (2):22–42, 2007.

[BV05] Laurent Bartholdi and Bálint Virág. Amenability via random walks. *Duke Math. J.*, 130(1):39–56, 2005. (available at http://arxiv.org/abs/math.GR/0305262).

[Ers04] Anna Erschler. Boundary behavior for groups of subexponential growth. *Annals of Math.*, 160(3):1183–1210, 2004.

[FG85] Jacek Fabrykowski and Narain Gupta. On groups with sub-exponential growth functions. *J. Indian Math. Soc. (N.S.)*, 49(3-4):249–256 (1987), 1985.

[GLSZ00] Rostislav I. Grigorchuk, Peter Linnell, Thomas Schick, and Andrzej Zuk. On a question of Atiyah. *C. R. Acad. Sci. Paris Sér. I Math.*, 331(9):663–668, 2000.

[GNS00] R. I. Grigorchuk, V. V. Nekrashevich, and V. I. Sushchanskiĭ. Automata, dynamical systems, and groups. *Tr. Mat. Inst. Steklova*, 231(Din. Sist., Avtom. i Beskon. Gruppy):134–214, 2000.

[Gri80] R. I. Grigorčuk. On Burnside's problem on periodic groups. *Funktsional. Anal. i Prilozhen.*, 14(1):53–54, 1980.

[Gri84] R. I. Grigorchuk. Degrees of growth of finitely generated groups and the theory of invariant means. *Izv. Akad. Nauk SSSR Ser. Mat.*, 48(5):939–985, 1984.

[Gri05] Rostislav Grigorchuk. Solved and unsolved problems around one group. In *Infinite groups: geometric, combinatorial and dynamical aspects*, volume 248 of *Progr. Math.*, pages 117–218. Birkhäuser, Basel, 2005.

[GS83] Narain Gupta and Saïd Sidki. On the Burnside problem for periodic groups. *Math. Z.*, 182(3):385–388, 1983.

[GŠ06] Rostislav Grigorchuk and Zoran Šuniḱ. Asymptotic aspects of Schreier graphs and Hanoi Towers groups. *C. R. Math. Acad. Sci. Paris*, 342(8):545–550, 2006.

[GŠ08] Rostislav Grigorchuk and Zoran Šunić. Schreier spectrum of the Hanoi Towers group on three pegs. In *Analysis on graphs and its applications*, volume 77 of *Proc. Sympos. Pure Math.*, pages 183–198. Amer. Math. Soc., Providence, RI, 2008.

[GS14] Rostislav Grigorchuk and Dmytro Savchuk. Self-similar groups acting essentially freely on the boundary of the binary rooted tree. In *Group Theory, Combinatorics, and Computing*, volume 611 of *Contemp. Math.* Amer. Math. Soc., Providence, RI, 2014.

[GSŠ07] Rostislav Grigorchuk, Dmytro Savchuk, and Zoran Šunić. The spectral problem, substitutions and iterated monodromy. *CRM Proceedings and Lecture Notes*, 42(8):225–248, 2007.

[GZ02a] Rostislav I. Grigorchuk and Andrzej Zuk. On a torsion-free weakly branch group defined by a three state automaton. *Internat. J. Algebra Comput.*, 12(1-2):223–246, 2002.

[GZ02b] Rostislav I. Grigorchuk and Andrzej Zuk. Spectral properties of a torsion-free weakly branch group defined by a three state automaton. In *Computational and statistical group theory (Las Vegas, NV/Hoboken, NJ, 2001)*, volume 298 of *Contemp. Math.*, pages 57–82. Amer. Math. Soc., Providence, RI, 2002.

[Kli13] Ines Klimann. The finiteness of a group generated by a 2-letter invertible-reversible Mealy automaton is decidable. In Natacha Portier and Thomas Wilke, editors, *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, volume 20 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 502–513, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Nek07] Volodymyr Nekrashevych. A minimal Cantor set in the space of 3-generated groups. *Geom. Dedicata*, 124:153–190, 2007.

[Sid00] Said Sidki. Automorphisms of one-rooted trees: growth, circuit structure, and acyclicity. *J. Math. Sci. (New York)*, 100(1):1925–1943, 2000.

[Sus79] V. I. Sushchansky. Periodic permutation p-groups and the unrestricted Burnside problem. *DAN SSSR.*, 247(3):557–562, 1979. (in Russian).

[VV07] Mariya Vorobets and Yaroslav Vorobets. On a free group of transformations defined by an automaton. *Geom. Dedicata*, 124:237–249, 2007.

[Wil04] John S. Wilson. On exponential growth and uniformly exponential growth for groups. *Invent. Math.*, 155(2):287–303, 2004.