YAGS
Yet Another Graph System
GAP4 Package

Version 0.8
by

R. Mac Kinney Romero[1]
M. A. Pizaña[1]
R. Villarroel-Flores[2]

[1]Departamento de Ingeniería Eléctrica
Universidad Autónoma Metropolitana
{rene,map}@xanum.uam.mx

[2]Centro de Investigación en Matemáticas
Universidad Autónoma del Estado de Hidalgo
rafaelv@uaeh.edu.mx

Partially supported by SEP-CONACyT, grant 183210.

July 2014

Contents

1 Basics

YAGS (Yet Another Graph System) is a system designed to aid in the study of graphs. Therefore it provides functions designed to help researchers in this field. The main goal was, as a start, to be thorough and provide as much functionality as possible, and at a later stage to increase the efficiency of the system. Furthermore, a module on genetic algorithms is provided to allow experiments with graphs to be carried out.

This chapter is intended as a gentle tutorial on working with YAGS (some knowledge of GAP and the basic use of a computer are assumed).

The tutorial is divided as follows:

- Using YAGS
- Definition of a graph
- A taxonomy of graphs
- Creating graphs
- Transforming graphs
- Experimenting on graphs

1.1 Using YAGS

YAGS is a GAP package an as such the *RequirePackage* directive is used to start YAGS

```
gap> RequirePackage("YAGS");

Loading  YAGS 0.01  (Yet Another Graph System),
by  R. MacKinney and M.A. Pizana
rene@xamanek.uam.mx, map@xamanek.uam.mx

true
```

a double semicolon can be used to avoid the banner.

Once the package has been loaded help can be obtained at anytime using the GAP help facility. For instance get help on the function *RandomGraph*:

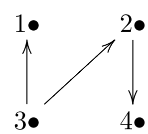
```
gap> ?RandomGraph
Help: Showing `yags: RandomGraph'

> RandomGraph( <n>, <p> )                                          F
> RandomGraph( <n> )                                               F

Returns  a  Random Graph of order <n>. The first form additionally takes a
parameter  <p>,  the probability of an edge to exist. A probability 1 will
return a Complete Graph and a probability 0 a Discrete Graph.
```

```
gap> RandomGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 4, 5 ], [ 4, 5 ], [  ], [ 1, 2, 5 ], [ 1, 2, 4 ] ] )
```

1.2 Definition of graphs

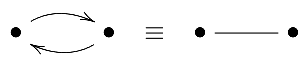
A graph is defined as follows. A graph G is a set of vertices V and a set of edges (arrows) E, $G = \{V,E\}$. The set of edges is a set of tuples of vertices (v_i, v_j) that belong to $V, v_i, v_j \in V$ representing that v_i, v_j are adjacent.

For instance, $(\{1,2,3,4\}, \{(1,3),(2,4),(3,2)\})$ is a graph with four vertices such that vertices 1 and 2 are adjacent to vertex 3 and vertex 2 is adjacent to vertex 4. Visually this can be seen as



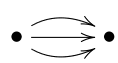
The adjacencies can also be represented as a matrix. This would be a boolean matrix M where two vertices i, j are adjacent if $M[i,j] = \mathit{true}$ and not adjacent otherwise.

Given two vertices i, j in graph G we will say that graph G has an **edge** $\{i,j\}$ if there is an arrow (i,j) and and arrow (j,i)·



If a graph G has an arrow that starts and finishes on the same vertex we say that graph G has a loop.

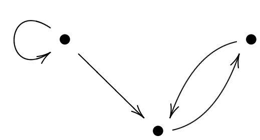
YAGS handles graphs that have arrows, edges and loops. Graphs that, for instance, have multiple arrows between vertices are not handled by YAGS .



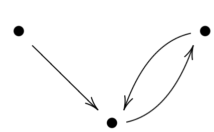
1.3 A taxonomy of graphs

There are several ways of characterizing graphs. YAGS uses a category system where any graph belongs to a specific category. The following is the list of graph categories in YAGS

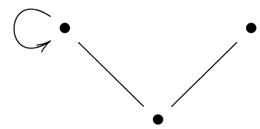
- *Graphs*: graphs with no particular property.



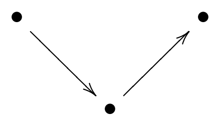
- *Loopless*: graphs with no loops.



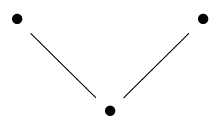
- *Undirected*: graphs with no arrows but only edges.



- *Oriented*: graphs with no edges but only arrows.



- *SimpleGraphs*: graphs with no loops and only edges.



The following figure shows the relationships among categories.

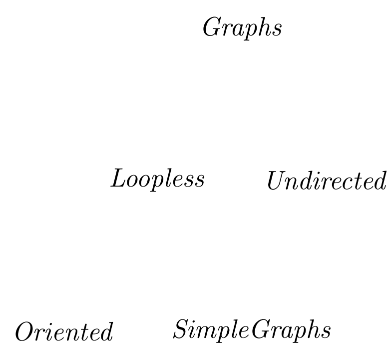


Figure 1: Graph Categories

YAGS uses the category of a graph to normalize it. This is helpful, for instance, when we define an undirected graph and inadvertently forget an arrow in its definition. The category of a graph can be given explicitly or implicitly. To do it explicitly the category must be given when creating a graph, as can be seen in the section 1.4. If no category is given the category is assumed to be the *DefaultCategory*. The default category can be changed at any time using the *SetDefaultCategory* function.

Further information regarding categories can be found on chapter 2.

1.4 Creating Graphs

There exist several ways to create a graph in YAGS. First, a GAP record can be used. To do so the record has to have either of

- Adjacency List
- Adjacency Matrix

in the graph presented in Section 1.2 the adjacency list would be

$$[[],[4],[1,2],[]]$$

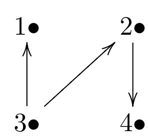
and the adjacency matrix

$$\begin{bmatrix} \mathit{false} & \mathit{false} & \mathit{false} & \mathit{false} \\ \mathit{false} & \mathit{false} & \mathit{false} & \mathit{true} \\ \mathit{true} & \mathit{true} & \mathit{false} & \mathit{false} \\ \mathit{false} & \mathit{false} & \mathit{false} & \mathit{false} \end{bmatrix}$$

To create a graph YAGS we also need the category the graph belongs to. We give this information to the *Graph* function. For instance to create the graph using the adjacency list we would use the following command:

```
gap> g:=Graph(rec(Category:=OrientedGraphs,Adjacencies:=[[],[4],[1,2],[]]));
Graph( Category := OrientedGraphs, Order := 4, Size := 3, Adjacencies :=
[ [  ], [ 4 ], [ 1, 2 ], [  ] ] )
```

This will create a graph g that represents the graph in Section 1.2.



Since the *DefaultCategory* is *SimpleGraphs* when YAGS starts up and the graph we have been using as an example is oriented we must explicitly give the category to YAGS. This is achieved using *Category:=OrientedGraphs* inside the record structure.

The same graph can be created using the function *GraphByAdjacencies* as in

```
gap> g:=GraphByAdjacencies([[],[4],[1,2],[]]:Category:=OrientedGraphs);
Graph( Category := OrientedGraphs, Order := 4, Size := 3, Adjacencies :=
[ [  ], [ 4 ], [ 1, 2 ], [  ] ] )
```

In this case to explicitly give the Category of the graph we use the construction *:Category:=OrientedGraphs* inside the function. This construction can be used in any function to explicitly give the category of a graph.

We said previously we can also use the adjacency matrix to create a graph. For instance the command

```
gap> g:=Graph(rec(Category:=OrientedGraphs,AdjMatrix:=
         [[false,false,false,false],[false,false,false,true],
         [true,true,false,false],[false,false,false,false]]));
Graph( Category := OrientedGraphs, Order := 4, Size := 3, Adjacencies :=
[ [  ], [ 4 ], [ 1, 2 ], [  ] ] )
```

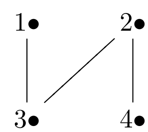
Creates the same graph. Note that we explicitly give the graph category as before. We also can use the command *AdjMatrix* as in

```
gap> g:=AdjMatrix(AdjMatrix:=[[false,false,false,false],
         [false,false,false,true],[true,true,false,false],
         [false,false,false,false]]):Category:=OrientedGraphs);
Graph( Category := OrientedGraphs, Order := 4, Size := 3, Adjacencies :=
[ [  ], [ 4 ], [ 1, 2 ], [  ] ] )
```

If we create the graph using any of the methods so far described omitting the graph category YAGS will create a graph normalized to the *DefaultCategory* which by default is *SimpleGraphs*

```
gap> g:=GraphByAdjacencies([[],[4],[1,2],[]];
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 3 ], [ 3, 4 ], [ 1, 2 ], [ 2 ] ] )
```

Which creates a graph with only edges

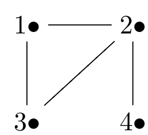


There are many functions to create graphs, some from existing graphs and some create interesting well known graphs.

Among the former we have the function *AddEdges* which adds edges to an existing graph

```
gap> g:=GraphByAdjacencies([[],[4],[1,2],[]]);
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 3 ], [ 3, 4 ], [ 1, 2 ], [ 2 ] ] )
gap> h:=AddEdges(g,[[1,2]]);
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 3 ], [ 1, 3, 4 ], [ 1, 2 ], [ 2 ] ] )
```

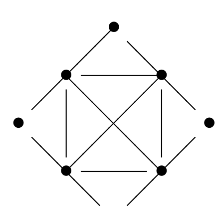
that yields the graph h



Among the latter we have the function *SunGraph* which takes an integer as argument and returns a fresh copy of a sun graph of the order given as argument.

```
gap> h:=SunGraph(4);
Graph( Category := SimpleGraphs, Order := 8, Size := 14, Adjacencies :=
[ [ 2, 8 ], [ 1, 3, 4, 6, 8 ], [ 2, 4 ], [ 2, 3, 5, 6, 8 ], [ 4, 6 ],
  [ 2, 4, 5, 7, 8 ], [ 6, 8 ], [ 1, 2, 4, 6, 7 ] ] )
```

that produces h as



Further information regarding constructing graphs can be found on chapter 3.

1.5 Transforming graphs

1.6 Experimenting on graphs

Coming soon!

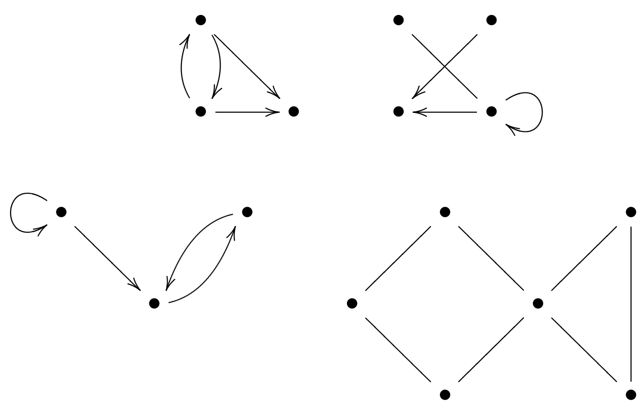
2 Categories

2.1 Graph Categories

1 ► `Graphs( )` C

Graphs are the base category used by YAGS. This category contains all graphs that can be represented in YAGS.

Among them we can find:

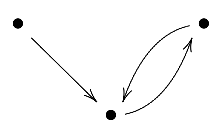


2 ► `LooplessGraphs( )` C

Loopless Graphs are graphs which have no loops.

A loop is an arrow that starts and finishes on the same vertex.

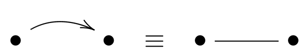
Loopless graphs have no such arrows.



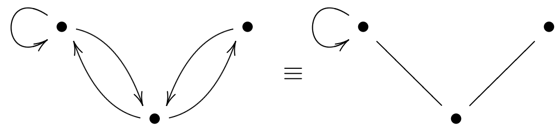
3 ► `UndirectedGraphs( )` C

Undirected Graphs are graphs which have no directed arrows.

Given two vertex i, j in graph G we will say that graph G has an **edge** $\{i, j\}$ if there is an arrow (i, j) and and arrow (j, i)·

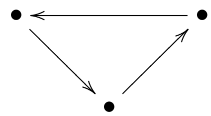


Undirected graphs have no arrows but only edges.



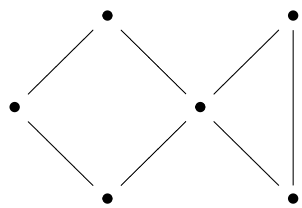
4 ▶ `OrientedGraphs( )` C

Oriented Graphs are graphs which have arrows in only one direction between any two vertices.
Oriented graphs have no edges but only arrows.



5 ▶ `SimpleGraphs( )` C

Simple Graphs are graphs with no loops and undirected.



The following figure shows the relationships among categories.

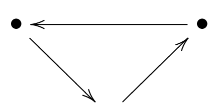
Graphs

Loopless *Undirected*

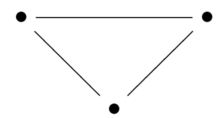
Oriented *SimpleGraphs*

Figure 2: Graph Categories

This relationship is important because when a graph is created it is normalized to the category it belongs. For instance, if we create a graph such as



as a simple graph YAGS will normalize the graph as



For further examples see the following section.

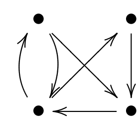
2.2 Default Category

There are several ways to specify the category in which a new graph will be created. There exists a *DefaultCategory* which tells YAGS to which category belongs any new graph by default. The *DefaultCategory* can be changed using the following function.

1 ▶ `SetDefaultGraphCategory(` C `)` F

Sets category C to be the default category for graphs. The default category is used, for instance, when constructing new graphs.

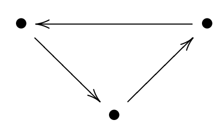
```
gap> SetDefaultGraphCategory(Graphs);
gap> g:=RandomGraph(4);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )
```



RandomGraph creates a random graphs belonging to the category graphs. The above graph has loops which are not permitted in simple graphs.

```
gap> SetDefaultGraphCategory(SimpleGraphs);
gap> g:=CopyGraph(g);
Graph( Category := SimpleGraphs, Order := 4, Size := 5, Adjacencies :=
[ [ 3, 4 ], [ 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
```

Now G is a simple graph.



In order to handle graphs with different categories there two functions available.

2 ▶ `GraphCategory( [`G`, ... ] )` F

Returns the minimal common category to a list of graphs. See Section 2 for the relationship among categories.

If the list is empty the default category is returned.

3 ▶ `TargetGraphCategory( [`G`, ... ] )` F

Returns the category which will be used to process a list of graphs. If an option category has been given it will return that category. Otherwise it will behave as Function *GraphCategory* (6). See Section 2 for the relationship among categories.

Finally we can test if a single graph belongs to a given category.

4 ▶ `in(` G`,` C `)` O

Returns `true` if graph G belongs to category C and `false` otherwise.

3 Constructing graphs

3.1 Primitives

The following functions create new graphs from a variety of sources.

1 ▶ `Graph( R )` O

Returns a new graph created from the record R. The record must provide the field *Category* and either the field *Adjacencies* or the field *AdjMatrix*

```
gap> Graph(rec(Category:=SimpleGraphs,Adjacencies:=[[2],[1]]));
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies := [ [ 2 ], [ 1 ] ] )
gap> Graph(rec(Category:=SimpleGraphs,AdjMatrix:=[[false, true],[true, false]]));
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies := [ [ 2 ], [ 1 ] ] )
```

Its main purpose is to import graphs from files, which could have been previously exported using `PrintTo`.

```
gap> g:=CycleGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
gap> PrintTo("aux.g","h1:=",g,";");
gap> Read("aux.g");
gap> h1;
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
```

–map

2 ▶ `GraphByAdjMatrix( M )` F

Returns a new graph created from an adjacency matrix M. The matrix M must be a square boolean matrix.

```
gap> m:=[ [ false, true, false ], [ true, false, true ], [ false, true, false ] ];;
gap> g:=GraphByAdjMatrix(m);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
gap> AdjMatrix(g);
[ [ false, true, false ], [ true, false, true ], [ false, true, false ] ]
```

Note, however, that the graph is forced to comply with the `TargetGraphCategory`.

```
gap> m:=[ [ true, true], [ false, false ] ];;
gap> g:=GraphByAdjMatrix(m);
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies := [ [ 2 ], [ 1 ] ] )
gap> AdjMatrix(g);
[ [ false, true ], [ true, false ] ]
```

–map

3 ► `GraphByAdjacencies(` A `)` F

Returns a new graph having A as its list of adjacencies. The order of the created graph is `Length(A)`, and the set of neighbors of vertex x is $A[x]$.

```
gap> GraphByAdjacencies([[2],[1,3],[2]]);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
```

Note, however, that the graph is forced to comply with the `TargetGraphCategory`.

```
gap> GraphByAdjacencies([[1,2,3],[],[]]);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2, 3 ], [ 1 ], [ 1 ] ] )
```

–map

4 ► `GraphByCompleteCover(` C `)` F

Returns the minimal graph where the elements of C are (the vertex sets of) complete subgraphs.

```
gap> GraphByCompleteCover([[1,2,3,4],[4,6,7]]);
Graph( Category := SimpleGraphs, Order := 7, Size := 9, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3, 6, 7 ], [  ], [ 4, 7 ],
  [ 4, 6 ] ] )
```

–map

5 ► `GraphByRelation(` V`,` R `)` F
► `GraphByRelation(` N`,` R `)` F

Returns a new graph created from a set of vertices V and a binary relation R, where $x \sim y$ iff $R(x, y) = \mathit{true}$. In the second form, N is an integer and V is assumed to be $\{1, 2, \ldots, N\}$.

```
gap> R:=function(x,y) return Intersection(x,y)<>[]; end;;
gap> GraphByRelation([[1,2,3],[3,4,5],[5,6,7]],R);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
gap> GraphByRelation(8,function(x,y) return AbsInt(x-y)<=2; end);
Graph( Category := SimpleGraphs, Order := 8, Size := 13, Adjacencies :=
[ [ 2, 3 ], [ 1, 3, 4 ], [ 1, 2, 4, 5 ], [ 2, 3, 5, 6 ], [ 3, 4, 6, 7 ],
  [ 4, 5, 7, 8 ], [ 5, 6, 8 ], [ 6, 7 ] ] )
```

–map

6 ► `GraphByWalks(` *walk1*`,` *walk2*`, ... )` F

Returns the minimal graph such that *walk1*, *walk2*, etc are walks.

```
gap> GraphByWalks([1,2,3,4,1],[1,5,6]);
Graph( Category := SimpleGraphs, Order := 6, Size := 6, Adjacencies :=
[ [ 2, 4, 5 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ], [ 1, 6 ], [ 5 ] ] )
```

Walks can be *nested*, which greatly improves the versatility of this function.

```
gap> GraphByWalks([1,[2,3,4],5],[5,6]);
Graph( Category := SimpleGraphs, Order := 6, Size := 9, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 5 ], [ 1, 2, 4, 5 ], [ 1, 3, 5 ], [ 2, 3, 4, 6 ], [ 5 ] ] )
```

–map

7 ▶ `IntersectionGraph(` L `)` F

Returns the intersection graph of the family of sets L. This graph has a vertex for every set in L, and two such vertices are adjacent iff the corresponding sets have non-empty intersection.

```
gap> IntersectionGraph([[1,2,3],[3,4,5],[5,6,7]]);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
```

–map

The following functions create graphs from existing graphs

8 ▶ `CopyGraph(` G `)` O

Returns a fresh copy of graph G. Only the order and adjacency information is copied, all other known attributes of G are not. Mainly used to transform a graph from one category to another. The new graph will be forced to comply with the `TargetGraphCategory`.

```
gap> g:=CompleteGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
gap> g1:=CopyGraph(g:GraphCategory:=OrientedGraphs);
Graph( Category := OrientedGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 3, 4 ], [ 4 ], [  ] ] )
gap> CopyGraph(g1:GraphCategory:=SimpleGraphs);
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
```

–map

9 ▶ `InducedSubgraph(` G, V `)` O

Returns the subgraph of graph G induced by the vertex set V.

```
gap> g:=CycleGraph(6);
Graph( Category := SimpleGraphs, Order := 6, Size := 6, Adjacencies :=
[ [ 2, 6 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4, 6 ], [ 1, 5 ] ] )
gap> InducedSubgraph(g,[3,4,6]);
Graph( Category := SimpleGraphs, Order := 3, Size := 1, Adjacencies :=
[ [ 2 ], [ 1 ], [  ] ] )
```

The order of the elements in V does matter.

```
gap> InducedSubgraph(g,[6,3,4]);
Graph( Category := SimpleGraphs, Order := 3, Size := 1, Adjacencies :=
[ [  ], [ 3 ], [ 2 ] ] )
```

–map

10 ▶ `RemoveVertices(` G, V `)` O

Returns a new graph created from graph G by removing the vertices in list V.

```
gap> g:=PathGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 4, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4 ] ] )
gap> RemoveVertices(g,[3]);
Graph( Category := SimpleGraphs, Order := 4, Size := 2, Adjacencies :=
[ [ 2 ], [ 1 ], [ 4 ], [ 3 ] ] )
gap> RemoveVertices(g,[1,3]);
Graph( Category := SimpleGraphs, Order := 3, Size := 1, Adjacencies :=
[ [  ], [ 3 ], [ 2 ] ] )
```

–map

11 ▶ `AddEdges(` G, E `)` O

Returns a new graph created from graph G by adding the edges in list E.

```
gap> g:=CycleGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
gap> AddEdges(g,[[1,3]]);
Graph( Category := SimpleGraphs, Order := 4, Size := 5, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 3 ] ] )
gap> AddEdges(g,[[1,3],[2,4]]);
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
```

–map

12 ▶ `RemoveEdges(` G, E `)` O

Returns a new graph created from graph G by removing the edges in list E.

```
gap> g:=CompleteGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
gap> RemoveEdges(g,[[1,2]]);
Graph( Category := SimpleGraphs, Order := 4, Size := 5, Adjacencies :=
[ [ 3, 4 ], [ 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
gap> RemoveEdges(g,[[1,2],[3,4]]);
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 3, 4 ], [ 3, 4 ], [ 1, 2 ], [ 1, 2 ] ] )
```

–map

13 ▶ `CliqueGraph(` G `)` A
▶ `CliqueGraph(` G, m `)` O

Returns the intersection graph of all the (maximal) cliques of G.

The additional parameter m aborts the computation when m cliques are found, even if they are all the cliques of G. If the bound m is reached, *fail* is returned.

```
gap> CliqueGraph(Octahedron);
Graph( Category := SimpleGraphs, Order := 8, Size := 24, Adjacencies :=
[ [ 2, 3, 4, 5, 6, 7 ], [ 1, 3, 4, 5, 6, 8 ], [ 1, 2, 4, 5, 7, 8 ],
  [ 1, 2, 3, 6, 7, 8 ], [ 1, 2, 3, 6, 7, 8 ], [ 1, 2, 4, 5, 7, 8 ],
  [ 1, 3, 4, 5, 6, 8 ], [ 2, 3, 4, 5, 6, 7 ] ] )
gap> CliqueGraph(Octahedron,9);
Graph( Category := SimpleGraphs, Order := 8, Size := 24, Adjacencies :=
[ [ 2, 3, 4, 5, 6, 7 ], [ 1, 3, 4, 5, 6, 8 ], [ 1, 2, 4, 5, 7, 8 ],
  [ 1, 2, 3, 6, 7, 8 ], [ 1, 2, 3, 6, 7, 8 ], [ 1, 2, 4, 5, 7, 8 ],
  [ 1, 3, 4, 5, 6, 8 ], [ 2, 3, 4, 5, 6, 7 ] ] )
gap> CliqueGraph(Octahedron,8);
fail
```

–map

3.2 Families

The following functions return well known graphs. Most of them can be found in Brandstadt, Le and Spinrad.

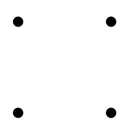
1 ► `DiscreteGraph(` *n* `)` F

Returns the discrete graph of order n. A discrete graph is a graph without edges.

```
gap> DiscreteGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 0, Adjacencies :=
[ [  ], [  ], [  ], [  ] ] )
```

–map

4-Discrete Graph



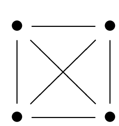
2 ► `CompleteGraph(` *n* `)` F

Returns the complete graph of order n. A complete graph is a graph where all vertices are connected to each other.

```
gap> CompleteGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
```

–map

4-Complete Graph



3 ► `PathGraph(` *n* `)` F

Returns the path graph on n vertices.

```
gap> PathGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3 ] ] )
```

–map

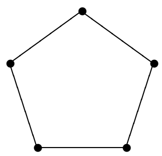
4-Path Graph

4 ► `CycleGraph( n )` F

Returns the cyclic graph on n vertices.

```
gap> CycleGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 2, 5 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 1, 4 ] ] )
```

–map

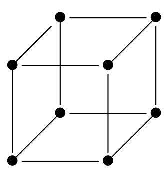
5-Cycle Graph 

5 ► `CubeGraph( n )` F

Returns the hypercube of dimension n. This is the box product (cartesian product) of n copies of K_2 (an edge).

```
gap> CubeGraph(3);
Graph( Category := SimpleGraphs, Order := 8, Size := 12, Adjacencies :=
[ [ 2, 3, 5 ], [ 1, 4, 6 ], [ 1, 4, 7 ], [ 2, 3, 8 ], [ 1, 6, 7 ],
[ 2, 5, 8 ], [ 3, 5, 8 ], [ 4, 6, 7 ] ] )
```

-map

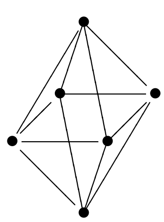
3-Cube Graph 

6 ► `OctahedralGraph( n )` F

Return the n-dimensional octahedron. This is the complement of n copies of K_2 (an edge). It is also the *(2n-2)*-regular graph on $2n$ vertices.

```
gap> OctahedralGraph(3);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 3, 4, 5, 6 ], [ 3, 4, 5, 6 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ],
[ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ] ] )
```

–map

3-Octahedral Graph 

7 ▶ `JohnsonGraph(` *n*, *r* `)` F

Returns the Johnson graph $J(n,r)\cdot$ A Johnson Graph is a graph constructed as follows. Each vertex represents a subset of the set $\{1,\ldots,n\}$ with cardinality $r\cdot$

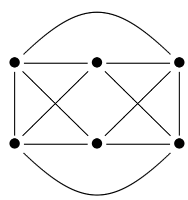
$$V(J(n,r)) = \{X \subset \{1,\ldots,n\} || X| = r\}$$

and there is an edge between two vertices if and only if the cardinality of the intersection of the sets they represent is $r-1$

$$X \sim X' \text{ iff } |X \cup X'| = r - 1\cdot$$

```
gap> JohnsonGraph(4,2);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 4, 6 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ],
  [ 1, 3, 4, 6 ], [ 2, 3, 4, 5 ] ] )
```

–map

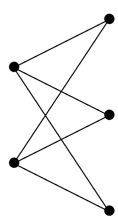
4,2-Johnson Graph 

8 ▶ `CompleteBipartiteGraph(` *n*, *m* `)` F

Returns the complete bipartite whose parts have order n and m respectively. This is the joint (Zykov sum) of two discrete graphs of order n and m.

```
gap> CompleteBipartiteGraph(2,3);
Graph( Category := SimpleGraphs, Order := 5, Size := 6, Adjacencies :=
[ [ 3, 4, 5 ], [ 3, 4, 5 ], [ 1, 2 ], [ 1, 2 ], [ 1, 2 ] ] )
```

–map

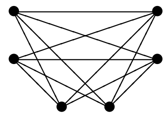
2,3-Complete Bipartite Graph 

9 ▶ `CompleteMultipartiteGraph(` *n1*, *n2* `[,` *n3* `...] )` F

Returns the complete multipartite graph where the orders of the parts are *n1*, *n2*, ... It is also the Zykov sum of discrete graphs of order *n1*, *n2*, ...

```
gap> CompleteMultipartiteGraph(2,2,2);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 3, 4, 5, 6 ], [ 3, 4, 5, 6 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ],
  [ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ] ] )
```

–map

2,2,2-Complete Multipartite Graph 

10 ▶ `RandomGraph(` n, p `)` F
▶ `RandomGraph(` n `)` F

Returns a random graph of order n taking the rational $p \in [0, 1]$ as the edge probability.

```
gap> RandomGraph(5,1/3);
Graph( Category := SimpleGraphs, Order := 5, Size := 2, Adjacencies :=
[ [ 5 ], [ 5 ], [  ], [  ], [ 1, 2 ] ] )
gap> RandomGraph(5,2/3);
Graph( Category := SimpleGraphs, Order := 5, Size := 6, Adjacencies :=
[ [ 4, 5 ], [ 3, 4, 5 ], [ 2, 4 ], [ 1, 2, 3 ], [ 1, 2 ] ] )
gap> RandomGraph(5,1/2);
Graph( Category := SimpleGraphs, Order := 5, Size := 4, Adjacencies :=
[ [ 2, 5 ], [ 1, 3, 5 ], [ 2 ], [  ], [ 1, 2 ] ] )
```

If p is ommited, the edge probability is taken to be 1/2.

```
gap> RandomGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 2, 3 ], [ 1 ], [ 1, 4, 5 ], [ 3, 5 ], [ 3, 4 ] ] )
gap> RandomGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 3, Adjacencies :=
[ [ 2, 5 ], [ 1, 4 ], [  ], [ 2 ], [ 1 ] ] )
```

–map

5-Random Graph

11 ▶ `WheelGraph(` N `)` O
▶ `WheelGraph(` N, *Radius* `)` O

In its first form `WheelGraph` returns the wheel graph on *N+1* vertices. This is the cone of a cycle: a central vertex adjacent to all the vertices of an N-cycle

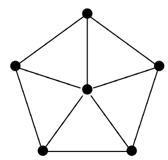
```
WheelGraph(5);
gap> Graph( Category := SimpleGraphs, Order := 6, Size := 10, Adjacencies :=
[ [ 2, 3, 4, 5, 6 ], [ 1, 3, 6 ], [ 1, 2, 4 ], [ 1, 3, 5 ], [ 1, 4, 6 ],
  [ 1, 2, 5 ] ] )
```

In its second form, `WheelGraph` returns returns the wheel graph, but adding *Radius-1* layers, each layer is a new N-cycle joined to the previous layer by a zigzagging *2N*-cycle. This graph is a triangulation of the disk.

```
gap> WheelGraph(5,2);
Graph( Category := SimpleGraphs, Order := 11, Size := 25, Adjacencies :=
[ [ 2, 3, 4, 5, 6 ], [ 1, 3, 6, 7, 8 ], [ 1, 2, 4, 8, 9 ], [ 1, 3, 5, 9, 10 ],
  [ 1, 4, 6, 10, 11 ], [ 1, 2, 5, 7, 11 ], [ 2, 6, 8, 11 ], [ 2, 3, 7, 9 ],
  [ 3, 4, 8, 10 ], [ 4, 5, 9, 11 ], [ 5, 6, 7, 10 ] ] )
gap> WheelGraph(5,3);
Graph( Category := SimpleGraphs, Order := 16, Size := 40, Adjacencies :=
[ [ 2, 3, 4, 5, 6 ], [ 1, 3, 6, 7, 8 ], [ 1, 2, 4, 8, 9 ], [ 1, 3, 5, 9, 10 ],
```

```
[ 1, 4, 6, 10, 11 ], [ 1, 2, 5, 7, 11 ], [ 2, 6, 8, 11, 12, 13 ],
[ 2, 3, 7, 9, 13, 14 ], [ 3, 4, 8, 10, 14, 15 ], [ 4, 5, 9, 11, 15, 16 ],
[ 5, 6, 7, 10, 12, 16 ], [ 7, 11, 13, 16 ], [ 7, 8, 12, 14 ],
[ 8, 9, 13, 15 ], [ 9, 10, 14, 16 ], [ 10, 11, 12, 15 ] ] )
```

–map

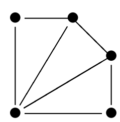
Wheel Graph of Order 5 

12 ▶ `FanGraph(` N `)` F

Returns the N-Fan: The join of a vertex and a *(N+1)*-path.

```
gap> FanGraph(4);
Graph( Category := SimpleGraphs, Order := 6, Size := 9, Adjacencies :=
[ [ 2, 3, 4, 5, 6 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 3, 5 ], [ 1, 4, 6 ],
[ 1, 5 ] ] )
```

–map

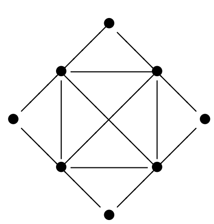
4-Fan Graph 

13 ▶ `SunGraph(` N `)` F

Returns the N-Sun: A complete graph on N vertices, K_N, with a corona made with a zigzagging *2N*-cycle glued to a N-cyle of the K_N.

```
gap> SunGraph(3);
Graph( Category := SimpleGraphs, Order := 6, Size := 9, Adjacencies :=
[ [ 2, 6 ], [ 1, 3, 4, 6 ], [ 2, 4 ], [ 2, 3, 5, 6 ], [ 4, 6 ],
  [ 1, 2, 4, 5 ] ] )
gap> SunGraph(4);
Graph( Category := SimpleGraphs, Order := 8, Size := 14, Adjacencies :=
[ [ 2, 8 ], [ 1, 3, 4, 6, 8 ], [ 2, 4 ], [ 2, 3, 5, 6, 8 ], [ 4, 6 ],
  [ 2, 4, 5, 7, 8 ], [ 6, 8 ], [ 1, 2, 4, 6, 7 ] ] )
```

–map

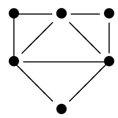
4-Sun Graph 

14 ▶ `SpikyGraph(` N `)` F

The spiky graph is constructed as follows: Take complete graph on N vertices, K_N, and then, for each the N subsets of *Vertices*(K_n) of order *N-1*, add an additional vertex which is adjacent precisely to this subset of *Vertices*(K_n).

```
gap> SpikyGraph(3);
Graph( Category := SimpleGraphs, Order := 6, Size := 9, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 4, 6 ], [ 1, 2, 5, 6 ], [ 1, 2 ], [ 1, 3 ],
  [ 2, 3 ] ] )
```

–map

3-Spiky Graph 

15 ▶ TrivialGraph V

The one vertex graph.

```
gap> TrivialGraph;
Graph( Category := SimpleGraphs, Order := 1, Size := 0, Adjacencies :=
[ [  ] ] )
```

–map

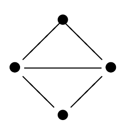
Trivial Graph

16 ▶ DiamondGraph V

The graph on 4 vertices and 5 edges.

```
gap> DiamondGraph;
Graph( Category := SimpleGraphs, Order := 4, Size := 5, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 3 ] ] )
```

–map

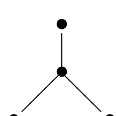
Diamond Graph 

17 ▶ ClawGraph V

The graph on 4 vertices, 3 edges, and maximum degree 3.

```
gap> ClawGraph;
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2, 3, 4 ], [ 1 ], [ 1 ], [ 1 ] ] )
```

–map

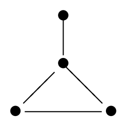
Claw Graph 

18 ▶ `PawGraph` V

The graph on 4 vertices, 4 edges and maximum degree 3: A triangle with a pendant vertex.

```
gap> PawGraph;
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2 ], [ 1, 3, 4 ], [ 2, 4 ], [ 2, 3 ] ] )
```

–map

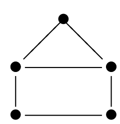
Paw Graph 

19 ▶ `HouseGraph` V

A 4-Cycle and a triangle glued by an edge.

```
gap> HouseGraph;
Graph( Category := SimpleGraphs, Order := 5, Size := 6, Adjacencies :=
[ [ 2, 4, 5 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3, 5 ], [ 1, 4 ] ] )
```

–map

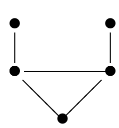
House Graph 

20 ▶ `BullGraph` V

A triangle with two pendant vertices (horns).

```
gap> BullGraph;
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 2 ], [ 1, 3, 4 ], [ 2, 4 ], [ 2, 3, 5 ], [ 4 ] ] )
```

–map

Bull Graph 

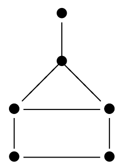
21 ▶ `AntennaGraph` V

A `HouseGraph` with a pendant vertex (antenna) on the roof.

```
gap> AntennaGraph;
Graph( Category := SimpleGraphs, Order := 6, Size := 7, Adjacencies :=
[ [ 2, 4, 5 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3, 5 ], [ 1, 4, 6 ], [ 5 ] ] )
```

–map

Antenna Graph



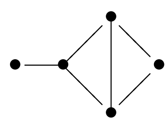
22 ► `KiteGraph` V

A diamond with a pending vertex and maximum degree 3.

```
gap> KiteGraph;
Graph( Category := SimpleGraphs, Order := 5, Size := 6, Adjacencies :=
[ [ 2 ], [ 1, 3, 4 ], [ 2, 4, 5 ], [ 2, 3, 5 ], [ 3, 4 ] ] )
```

–map

Kite Graph



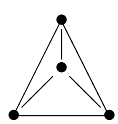
23 ► `Tetrahedron` V

The 1-skeleton of Plato's tetrahedron.

```
gap> Tetrahedron;
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
```

–map

Tetrahedron



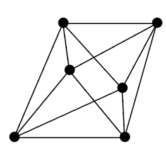
24 ► `Octahedron` V

The 1-skeleton of Plato's octahedron.

```
gap> Octahedron;
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 3, 4, 5, 6 ], [ 3, 4, 5, 6 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ],
  [ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ] ] )
```

–map

Octahedron



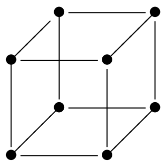
25 ► `Cube` V

The 1-skeleton of Plato's cube.

```
gap> Cube;
Graph( Category := SimpleGraphs, Order := 8, Size := 12, Adjacencies :=
[ [ 2, 3, 5 ], [ 1, 4, 6 ], [ 1, 4, 7 ], [ 2, 3, 8 ], [ 1, 6, 7 ],
  [ 2, 5, 8 ], [ 3, 5, 8 ], [ 4, 6, 7 ] ] )
```

–map

Cube Graph



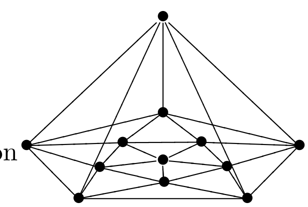
26 ► `Icosahedron` V

The 1-skeleton of Plato's icosahedron.

```
gap> Icosahedron;
Graph( Category := SimpleGraphs, Order := 12, Size := 30, Adjacencies :=
[ [ 2, 3, 4, 5, 6 ], [ 1, 3, 6, 9, 10 ], [ 1, 2, 4, 10, 11 ],
    [ 1, 3, 5, 7, 11 ], [ 1, 4, 6, 7, 8 ], [ 1, 2, 5, 8, 9 ],
    [ 4, 5, 8, 11, 12 ], [ 5, 6, 7, 9, 12 ], [ 2, 6, 8, 10, 12 ],
    [ 2, 3, 9, 11, 12 ], [ 3, 4, 7, 10, 12 ], [ 7, 8, 9, 10, 11 ] ] )
```

–map

Icosahedron



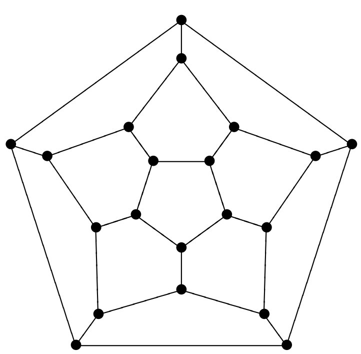
27 ► `Dodecahedron` V

The 1-skeleton of Plato's Dodecahedron.

gap¿ Dodecahedron; Graph(Category := SimpleGraphs, Order := 20, Size := 30, Adjacencies := [[2, 5, 6], [1, 3, 7], [2, 4, 8], [3, 5, 9], [1, 4, 10], [1, 11, 15], [2, 11, 12], [3, 12, 13], [4, 13, 14], [5, 14, 15], [6, 7, 16], [7, 8, 17], [8, 9, 18], [9, 10, 19], [6, 10, 20], [11, 17, 20], [12, 16, 18], [13, 17, 19], [14, 18, 20], [15, 16, 19]])

–map

Dodecahedron



3.3 Unary operations

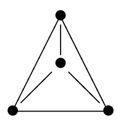
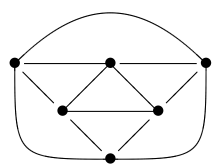
These are operations that can be performed over graphs.

1 ▶ `LineGraph( <G> )` O

Returns the line graph ¡L(G)¿ of graph ¡G¿. The line graph is the intersection graph of the edges of ¡G¿, ¡i.e.¿ the vertices of $L(G)$ are the edges of ¡G¿ two of them being adjacent iff they are incident.

```
gap> g:=Tetrahedron;
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
gap> LineGraph(g);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 4, 6 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ],
  [ 1, 3, 4, 6 ], [ 2, 3, 4, 5 ] ] )
```

–map

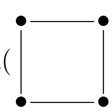
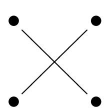
LineGraph() = 

2 ▶ `ComplementGraph( <G> )` A

Computes the complement of graph ¡G¿. The complement of a graph is created as follows: Create a graph ¡G'¿ with same vertices of ¡G¿. For each ¡x¿, ¡y¿ $\in$ ¡G¿ if ¡x¿ $\nsim$ ¡y¿ in ¡G¿ then ¡x¿ $\sim$ ¡y¿ in ¡G'¿

```
gap> g:=ClawGraph;
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2, 3, 4 ], [ 1 ], [ 1 ], [ 1 ] ] )
gap> ComplementGraph(g);
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [  ], [ 3, 4 ], [ 2, 4 ], [ 2, 3 ] ] )
```

–map

ComplementGraph() = 

3 ▶ `QuotientGraph( <G>, <P> )` O
▶ `QuotientGraph( <G>, <L1>, <L2> )` O

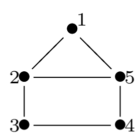
Returns the quotient graph of graph ¡G¿ given a vertex partition ¡P¿, by identifying any two vertices in the same part. The vertices of the quotient graph are the parts in the partition ¡P¿ two of them being adjacent iff any vertex in one part is adjacent to any vertex in the other part. Singletons may be omited in P.

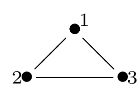
```
gap> g:=PathGraph(8);;
gap> QuotientGraph(g,[[1,5,8],[2],[3],[4],[6],[7]]);
Graph( Category := SimpleGraphs, Order := 6, Size := 7, Adjacencies :=
[ [ 2, 4, 5, 6 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ], [ 1, 6 ], [ 1, 5 ] ] )
gap> QuotientGraph(g,[[1,5,8]]);
Graph( Category := SimpleGraphs, Order := 6, Size := 7, Adjacencies :=
[ [ 2, 4, 5, 6 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ], [ 1, 6 ], [ 1, 5 ] ] )
```

In its second form, `QuotientGraph` identifies each vertex in list ¡L1¿, with the corresponding vertex in list ¡L2¿. ¡L1¿ and ¡L2¿ must have the same length, but any or both of them may have repetitions.

```
gap> g:=PathGraph(8);;
gap> QuotientGraph(g,[[1,7],[4,8]]);
Graph( Category := SimpleGraphs, Order := 6, Size := 7, Adjacencies :=
[ [ 2, 4, 6 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3, 5 ], [ 4, 6 ], [ 1, 5 ] ] )
gap> QuotientGraph(g,[1,4],[7,8]);
Graph( Category := SimpleGraphs, Order := 6, Size := 7, Adjacencies :=
[ [ 2, 4, 6 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3, 5 ], [ 4, 6 ], [ 1, 5 ] ] )
```

–map

QuotientGraph(

, [2, 3], [4, 5]) = 

3.4 Binary operations

These are binary operations that can be performed over graphs.

1 ► `BoxProduct(` G, H `)` O

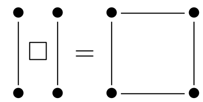
Returns the box product, $G \square H$, of two graphs G and H (also known as the cartesian product).

The box product is calculated as follows:

For each pair of vertices $g \in G, h \in H$ we create a vertex (g, h). Given two such vertices (g, h) and (g', h') they are adjacent *iff* $g = g'$ and $h \sim h'$ or $g \sim g'$ and $h = h'$.

```
gap> g1:=PathGraph(3);g2:=CycleGraph(4);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
gap> g1g2:=BoxProduct(g1,g2);
Graph( Category := SimpleGraphs, Order := 12, Size := 20, Adjacencies :=
[ [ 2, 4, 5 ], [ 1, 3, 6 ], [ 2, 4, 7 ], [ 1, 3, 8 ], [ 1, 6, 8, 9 ],
  [ 2, 5, 7, 10 ], [ 3, 6, 8, 11 ], [ 4, 5, 7, 12 ], [ 5, 10, 12 ],
  [ 6, 9, 11 ], [ 7, 10, 12 ], [ 8, 9, 11 ] ] )
gap> VertexNames(g1g2);
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 1 ], [ 2, 2 ], [ 2, 3 ],
  [ 2, 4 ], [ 3, 1 ], [ 3, 2 ], [ 3, 3 ], [ 3, 4 ] ]
```

–map



2 ▶ `TimesProduct( G, H )` O

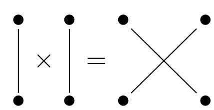
Returns the times product of two graphs G and H, $G \times H$ (also known as the tensor product).

The times product is computed as follows:

For each pair of vertices $g \in G, h \in H$ we create a vertex (g, h). Given two such vertices (g, h) and (g', h') they are adjacent *iff* $g \sim g'$ and $h \sim h'$.

```
gap> g1:=PathGraph(3);g2:=CycleGraph(4);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
gap> g1g2:=TimesProduct(g1,g2);
Graph( Category := SimpleGraphs, Order := 12, Size := 16, Adjacencies :=
[ [ 6, 8 ], [ 5, 7 ], [ 6, 8 ], [ 5, 7 ], [ 2, 4, 10, 12 ], [ 1, 3, 9, 11 ],
  [ 2, 4, 10, 12 ], [ 1, 3, 9, 11 ], [ 6, 8 ], [ 5, 7 ], [ 6, 8 ], [ 5, 7 ] ] )
gap> VertexNames(g1g2);
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 1 ], [ 2, 2 ], [ 2, 3 ],
  [ 2, 4 ], [ 3, 1 ], [ 3, 2 ], [ 3, 3 ], [ 3, 4 ] ]
```

–map



3 ▶ `BoxTimesProduct( G, H )` O

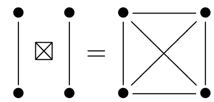
Returns the boxtimes product of two graphs G and H, $G \boxtimes H$ (also known as the strong product).

The box times product is calculated as follows:

For each pair of vertices $g \in G, h \in H$ we create a vertex (g, h). Given two such vertices (g, h) and (g', h') such that $(g, h) \neq (g', h')$ they are adjacent *iff* $g \simeq g'$ and $h \simeq h'$.

```
gap> g1:=PathGraph(3);g2:=CycleGraph(4);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
gap> g1g2:=BoxTimesProduct(g1,g2);
Graph( Category := SimpleGraphs, Order := 12, Size := 36, Adjacencies :=
[ [ 2, 4, 5, 6, 8 ], [ 1, 3, 5, 6, 7 ], [ 2, 4, 6, 7, 8 ], [ 1, 3, 5, 7, 8 ],
  [ 1, 2, 4, 6, 8, 9, 10, 12 ], [ 1, 2, 3, 5, 7, 9, 10, 11 ],
  [ 2, 3, 4, 6, 8, 10, 11, 12 ], [ 1, 3, 4, 5, 7, 9, 11, 12 ],
  [ 5, 6, 8, 10, 12 ], [ 5, 6, 7, 9, 11 ], [ 6, 7, 8, 10, 12 ],
  [ 5, 7, 8, 9, 11 ] ] )
gap> VertexNames(g1g2);
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 1 ], [ 2, 2 ], [ 2, 3 ],
  [ 2, 4 ], [ 3, 1 ], [ 3, 2 ], [ 3, 3 ], [ 3, 4 ] ]
```

–map



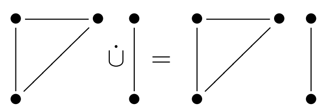
In the previous examples k^2 (*i.e.* the complete graph or order two) was chosen because it better pictures how the operators work.

4 ▶ `DisjointUnion( G, H )` O

Returns the disjoint union of two graphs G and H, $G \dot{\cup} H$.

```
gap> g1:=PathGraph(3);g2:=PathGraph(2);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies :=
[ [ 2 ], [ 1 ] ] )
gap> DisjointUnion(g1,g2);
Graph( Category := SimpleGraphs, Order := 5, Size := 3, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ], [ 5 ], [ 4 ] ] )
```

–map



5 ▶ `Join( G, H )` O

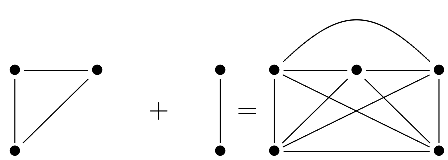
Returns the result of joining graph G and H, $G + H$ (also known as the Zykov sum).

Joining graphs is computed as follows:

First, we obtain the disjoint union of graphs G and H. Second, for each vertex $g \in G$ we add an edge to each vertex $h \in H$.

```
gap> g1:=DiscreteGraph(2);g2:=CycleGraph(4);
Graph( Category := SimpleGraphs, Order := 2, Size := 0, Adjacencies :=
[ [  ], [  ] ] )
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
gap> Join(g1,g2);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 3, 4, 5, 6 ], [ 3, 4, 5, 6 ], [ 1, 2, 4, 6 ], [ 1, 2, 3, 5 ],
  [ 1, 2, 4, 6 ], [ 1, 2, 3, 5 ] ] )
```

–map



6 ▶ `GraphSum( G, L )` O

Returns the lexicographic sum of a list of graphs L over a graph G.

The lexicographic sum is computed as follows:

Given G, with $Order(G) = n$ and a list of n graphs $L = [G_1, \ldots, G_n]$, We take the disjoint union of $G_1, G_2, \ldots, G_n$ and then we add all the edges between G_i and G_j whenever $[i, j]$ is and edge of G.

If L contains holes, the trivial graph is used in place.

```
gap> t:=TrivialGraph;; g:=CycleGraph(4);;
gap> GraphSum(PathGraph(3),[t,g,t]);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 5, 6 ], [ 1, 2, 4, 6 ], [ 1, 3, 5, 6 ],
  [ 1, 2, 4, 6 ], [ 2, 3, 4, 5 ] ] )
gap> GraphSum(PathGraph(3),[,g,]);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 5, 6 ], [ 1, 2, 4, 6 ], [ 1, 3, 5, 6 ],
  [ 1, 2, 4, 6 ], [ 2, 3, 4, 5 ] ] )
```

–map

7 ▶ `Composition(` G`,` H `)` O

Returns the composition $G[H]$ of two graphs G and H.

A composition of graphs is obtained by calculating the GraphSum of G with *Order(G)* copies of H, $G[H] = GraphSum(G, [H, \ldots, H])$.

```
gap> g1:=CycleGraph(4);;g2:=DiscreteGraph(2);;
gap> Composition(g1,g2);
Graph( Category := SimpleGraphs, Order := 8, Size := 16, Adjacencies :=
[ [ 3, 4, 7, 8 ], [ 3, 4, 7, 8 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ],
  [ 3, 4, 7, 8 ], [ 3, 4, 7, 8 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ] ] )
```

–map

4 Inspecting Graphs

4.1 Atributes and properties of graphs

The following are functions to obtain atributes and properties of graphs.

1 ► `AdjMatrix(` G `)` A

Returns the adjacency matrix of graph G.

```
gap> AdjMatrix(CycleGraph(4));
[ [ false, true, false, true ], [ true, false, true, false ],
  [ false, true, false, true ], [ true, false, true, false ] ]
```

–map

2 ► `Order(` G `)` A

Returns the number of vertices, of graph G.

```
gap> Order(Icosahedron);
12
```

–map

3 ► `Size(` G `)` A

Returns the number of edges of graph G.

```
gap> Size(Icosahedron);
30
```

–map

4 ► `VertexNames(` G `)` A

Return the list of names of the vertices of G. The vertices of a graph in YAGS are always $\{1, 2, \ldots, \mathit{Order}(G)\}$, but depending on how the graph was constructed, its vertices may have also some *names*, that help us identify the origin of the vertices. YAGS will always try to store meaninful names for the vertices. For example, in the case of the LineGraph, the vertex names of the new graph are the edges of the old graph.

```
gap> g:=LineGraph(DiamondGraph);
Graph( Category := SimpleGraphs, Order := 5, Size := 8, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4, 5 ], [ 1, 2, 5 ], [ 1, 2, 5 ], [ 2, 3, 4 ] ] )
gap> VertexNames(g);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 3, 4 ] ]
gap> Edges(DiamondGraph);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 3, 4 ] ]
```

–map

5 ► `IsCompleteGraph(` G `)` P
► `QtfyIsCompleteGraph(` G `)` A

Returns `true` if graph G is a complete graph, `false` otherwise. In a complete graph every pair of vertices is an edge.

–map

6 ► `IsLoopless(` G `)` P
► `QtfyIsLoopless(` G `)` A

Returns `true` if graph G have no loops, `false` otherwise. Loops are edges from a vertex to itself.

–map

7 ► `IsUndirected(` G `)` P
► `QtfyIsUndirected(` G `)` A

Returns `true` if graph G is an undirected graph, `false` otherwise. Regardless of the categories that G belongs to, G is undirected if whenever `[x,y]` is an edge of G, `[y,x]` is also an egde of G.

–map

8 ► `IsOriented(` G `)` P
► `QtfyIsOriented(` G `)` A

Returns `true` if graph G is an oriented graph, `false` otherwise. Regardless of the categories that G belongs to, G is oriented if whenever `[x,y]` is an edge of G, `[y,x]` is not.

–map

9 ► `CliqueNumber(` G `)` A

Returns the order, $\omega(G)$, of a maximum clique of G.

```
gap> g:=SunGraph(4);
Graph( Category := SimpleGraphs, Order := 8, Size := 14, Adjacencies :=
[ [ 2, 8 ], [ 1, 3, 4, 6, 8 ], [ 2, 4 ], [ 2, 3, 5, 6, 8 ], [ 4, 6 ],
  [ 2, 4, 5, 7, 8 ], [ 6, 8 ], [ 1, 2, 4, 6, 7 ] ] )
gap> CliqueNumber(g);
4
```

–map

10 ► `Cliques(` G `)` A
► `Cliques(` G, m `)` O

Returns the set of all (maximal) cliques of a graph G. A clique is a maximal complete subgraph. Here, we use the Bron-Kerbosch algorithm [BK73].

In the second form, It stops computing cliques after m of them have been found.

```
gap> Cliques(Octahedron);
[ [ 1, 3, 5 ], [ 1, 3, 6 ], [ 1, 4, 5 ], [ 1, 4, 6 ], [ 2, 3, 5 ],
  [ 2, 3, 6 ], [ 2, 4, 5 ], [ 2, 4, 6 ] ]
gap> Cliques(Octahedron,4);
[ [ 1, 3, 5 ], [ 1, 3, 6 ], [ 1, 4, 5 ], [ 1, 4, 6 ] ]
```

–map

11 ► `IsCliqueHelly(` G `)` A

Returns `true` if the set of (maximal) cliques G satisfy the *Helly* property.

The Helly property is defined as follows:

A non-empty family $\mathcal{F}$ of non-empty sets satisfies the Helly property if every pairwise intersecting subfamily of $\mathcal{F}$ has a non-empty total intersection.

Here we use the Dragan-Szwarcfiter characterization [Dra89,Szw97] to compute the Helly property.

```
gap> g:=SunGraph(3);
Graph( Category := SimpleGraphs, Order := 6, Size := 9, Adjacencies :=
[ [ 2, 6 ], [ 1, 3, 4, 6 ], [ 2, 4 ], [ 2, 3, 5, 6 ], [ 4, 6 ],
  [ 1, 2, 4, 5 ] ] )
gap> IsCliqueHelly(g);
false
```

–map

4.2 Information about graphs

The following functions give information regarding graphs.

1 ▶ `IsSimple(` G `)` O

Returns `true` if graph G is a simple graph, `false` otherwise. Regardless of the categories that G belongs to, G is simple if and only if G is undirected and loopless.

Returns `true` if the graph G is simple regardless of its category.

–map

2 ▶ `QtfyIsSimple(` G `)` O

Returns how far is graph G from being simple.

3 ▶ `Adjacency(` G, v `)` O

Returns the adjacency list of vertex v in G.

```
gap> g:=PathGraph(3);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
gap> Adjacency(g,1);
[ 2 ]
gap> Adjacency(g,2);
[ 1, 3 ]
```

–map

4 ▶ `Adjacencies(` G `)` O

Returns the adjacency lists of graph G.

```
gap> g:=PathGraph(3);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
gap> Adjacencies(g);
[ [ 2 ], [ 1, 3 ], [ 2 ] ]
```

–map

5 ▶ `VertexDegree(` G, v `)` O

Returns the degree of vertex v in Graph G.

```
gap> g:=PathGraph(3);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
gap> VertexDegree(g,1);
1
gap> VertexDegree(g,2);
2
```

–map

6 ▶ `VertexDegrees(` G `)` O

Returns the list of degrees of the vertices in graph G.

```
gap> g:=GemGraph;
Graph( Category := SimpleGraphs, Order := 5, Size := 7, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 3, 5 ], [ 1, 4 ] ] )
gap> VertexDegrees(g);
[ 4, 2, 3, 3, 2 ]
```

–map

7 ▶ `Edges(` G `)` O

Returns the list of edges of graph G.

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

–map

8 ▶ `CompletesOfGivenOrder(` G`,` o `)` O

This operation finds all complete subgraphs of order o in graph G.

```
gap> g:=SunGraph(4);
Graph( Category := SimpleGraphs, Order := 8, Size := 14, Adjacencies :=
[ [ 2, 8 ], [ 1, 3, 4, 6, 8 ], [ 2, 4 ], [ 2, 3, 5, 6, 8 ], [ 4, 6 ],
  [ 2, 4, 5, 7, 8 ], [ 6, 8 ], [ 1, 2, 4, 6, 7 ] ] )
gap> CompletesOfGivenOrder(g,3);
[ [ 1, 2, 8 ], [ 2, 3, 4 ], [ 2, 4, 6 ], [ 2, 4, 8 ], [ 2, 6, 8 ],
  [ 4, 5, 6 ], [ 4, 6, 8 ], [ 6, 7, 8 ] ]
gap> CompletesOfGivenOrder(g,4);
[ [ 2, 4, 6, 8 ] ]
```

–map

4.3 Distances

These are functions that measure distances between graphs.

1 ▶ `Distance(` G`,` x`,` y `)` O

Returns the length of a minimal path connecting x to y in G.

```
gap> Distance(CycleGraph(5),1,3);
2
gap> Distance(CycleGraph(5),1,5);
1
```

–map

2 ▶ `DistanceMatrix(` G `)` A

Returns the distance matrix D of a graph G: D[x][y] is the distance in G from vertex x to vertex y. The matrix may be asymmetric if the graph is not simple. An infinite entry in the matrix means that there is no path between the vertices. Floyd's algorithm is used to compute the matrix.

```
gap> g:=PathGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3 ] ] )
gap> Display(DistanceMatrix(g));
[ [  0,  1,  2,  3 ],
  [  1,  0,  1,  2 ],
  [  2,  1,  0,  1 ],
  [  3,  2,  1,  0 ] ]
gap> g:=PathGraph(4:GraphCategory:=OrientedGraphs);
Graph( Category := OrientedGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2 ], [ 3 ], [ 4 ], [  ] ] )
gap> Display(DistanceMatrix(g));
[ [         0,         1,         2,         3 ],
  [  infinity,         0,         1,         2 ],
  [  infinity,  infinity,         0,         1 ],
  [  infinity,  infinity,  infinity,         0 ] ]
```

–map

3 ▶ `Diameter(` G `)` A

Returns the maximum among the distances between pairs of vertices of G.

```
gap> g:=CycleGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 2, 5 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 1, 4 ] ] )
gap> Diameter(g);
2
```

–map

4 ▶ `Excentricity(` G, x `)` F

Returns the distance from a vertex x in graph G to its most distant vertex in G.

```
gap> g:=PathGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 4, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4 ] ] )
gap> Excentricity(g,1);
4
gap> Excentricity(g,3);
2
```

–map

5 ▶ `Radius(` G `)` A

Returns the minimal excentricity among the vertices of graph G.

```
gap> Radius(PathGraph(5));
2
```

–map

6 ▶ `Distances(` G, A, B `)` O

Given two lists of vertices A, B of a graph G, `Distances` returns the list of distances for every pair in the cartesian product of A and B. The order of the vertices in lists A and B affects the order of the list of distances returned.

```
gap> g:=CycleGraph(5);;
gap> Distances(g, [1,3], [2,4]);
[ 1, 2, 1, 1 ]
gap> Distances(g, [3,1], [2,4]);
[ 1, 1, 1, 2 ]
```

–map

7 ▶ `DistanceSet(` G, A, B `)` O

Given two subsets of vertices A, B of a graph G, `DistanceSet` returns the set of distances for every pair in the cartesian product of A and B.

```
gap> g:=CycleGraph(5);;
gap> DistanceSet(g, [1,3], [2,4]);
[ 1, 2 ]
```

–map

8 ▶ `DistanceGraph(` G, D `)` O

Given a graph G and list of distances D, `DistanceGraph` returns the new graph constructed on the vertices of G where two vertices are adjacent iff the distance (in G) between them belongs to the list D.

```
gap> g:=CycleGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 2, 5 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 1, 4 ] ] )
gap> DistanceGraph(g,[2]);
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 3, 4 ], [ 4, 5 ], [ 1, 5 ], [ 1, 2 ], [ 2, 3 ] ] )
gap> DistanceGraph(g,[1,2]);
Graph( Category := SimpleGraphs, Order := 5, Size := 10, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 4, 5 ], [ 1, 2, 4, 5 ], [ 1, 2, 3, 5 ],
  [ 1, 2, 3, 4 ] ] )
```

–map

9 ▶ `PowerGraph(` G, e `)` O

Returns the `DistanceGraph` of G using `[0, 1, ...,` e`]` as the list of distances. Note that the distance `0` in the list produces loops in the new graph only when the `TargetGraphCategory` admits loops.

```
gap> g:=PathGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 4, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4 ] ] )
gap> PowerGraph(g,1);
Graph( Category := SimpleGraphs, Order := 5, Size := 4, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4 ] ] )
gap> PowerGraph(g,1:GraphCategory:=Graphs);
Graph( Category := Graphs, Order := 5, Size := 13, Adjacencies :=
[ [ 1, 2 ], [ 1, 2, 3 ], [ 2, 3, 4 ], [ 3, 4, 5 ], [ 4, 5 ] ] )
```

–map

5 Morphisms of Graphs

There exists several classes of morphisms that can be found on graphs. Moreover, sometimes we want to find a combination of them. For this reason YAGS uses a unique mechanism for dealing with morphisms. This mechanisms allows to find any combination of morphisms using three underlying operations.

5.1 Core Operations

The following operations do all the work of finding morphisms that comply with all the properties given in a list. The list of checks that each function receives can have any of the following elements.

- CHQ_METRIC *Metric*
- CHQ_MONO *Mono*
- CHQ_FULL *Full*
- CHQ_EPI *Epi*
- CHQ_CMPLT *Complete*
- CHQ_ISO *Iso*

Additionally it must have at least one of the following.

- CHQ_WEAK *Weak*
- CHQ_MORPH *Morph*

These properties are detailed in the next section.

1 ► `PropertyMorphism(` *G1* `,` *G2* `,` *c* `)` O

Returns the first morphisms that is true for the list of checks *c* given graphs *G1* and *G2*.

```
gap> PropertyMorphism(CycleGraph(4),CompleteGraph(4),[CHQ_MONO,CHQ_MORPH]);
[ 1, 2, 3, 4 ]
```

2 ► `PropertyMorphisms(` *G1* `,` *G2* `,` *c* `)` O

Returns all morphisms that are true for the list of checks *c* given graphs *G1* and *G2*.

```
gap> PropertyMorphism(CycleGraph(4),CompleteGraph(4),[CHQ_MONO,CHQ_MORPH]);
[ [ 1, 2, 3, 4 ], [ 1, 2, 4, 3 ], [ 1, 3, 2, 4 ], [ 1, 3, 4, 2 ],
  [ 1, 4, 2, 3 ], [ 1, 4, 3, 2 ], [ 2, 1, 3, 4 ], [ 2, 1, 4, 3 ],
  [ 2, 3, 1, 4 ], [ 2, 3, 4, 1 ], [ 2, 4, 1, 3 ], [ 2, 4, 3, 1 ],
  [ 3, 1, 2, 4 ], [ 3, 1, 4, 2 ], [ 3, 2, 1, 4 ], [ 3, 2, 4, 1 ],
  [ 3, 4, 1, 2 ], [ 3, 4, 2, 1 ], [ 4, 1, 2, 3 ], [ 4, 1, 3, 2 ],
  [ 4, 2, 1, 3 ], [ 4, 2, 3, 1 ], [ 4, 3, 1, 2 ], [ 4, 3, 2, 1 ] ]
```

3 ► `NextPropertyMorphism(` *G1* `,` *G2* `,` *m* `,` *c* `)` O

Returns the next morphisms that is true for the list of checks *c* given graphs *G1* and *G2* starting with (possibly incomplete) morphism *m*. Note that if *m* is a variable the operation will change its value to the result of the operation.

```
gap> f:=[];;
gap> NextPropertyMorphism(CycleGraph(4),CompleteGraph(4),f,[CHQ_MONO,CHQ_MORPH$
[ 1, 2, 3, 4 ]
gap> NextPropertyMorphism(CycleGraph(4),CompleteGraph(4),f,[CHQ_MONO,CHQ_MORPH$
[ 1, 2, 4, 3 ]
gap> f;
[ 1, 2, 4, 3 ]
```

5.2 Morphisms

For all the definitions we assume we have a morphism $\varphi : G \to H\cdot$ The properties for creating morphisms are the following:

Metric A morphism is metric if the distance (see section 6) of any two vertices remains constant

$$d_G(x, y) = d_H(\varphi(x), \varphi(y))\cdot$$

Mono A morphism is mono if two different vertices in G map to two different vertices in H

$$x \neq y \implies \varphi(x) \neq \varphi(y)\cdot$$

Full A morphism is full if every edge in G is mapped to an edge in H.

$$|H| = |G|\cdot$$

Not yet implemented.

Epi A morphism is Epi if for each vertex in H exist a vertex in G that is mapped from.

$$\forall x \in H \exists x_0 \in G \bullet \varphi(x_0) = x$$

Complete A morphism is complete iff the inverse image of any complete of H is a complete of $G\cdot$

Iso An isomorphism is a bimorphism which is also complete.

Aditionally they must be one of the following

Weak A morphism is weak if x adjacent to y in G means their mappings are adjacent in H

$$x, y \in G \wedge x \simeq y \Rightarrow \varphi(x) \simeq \varphi(y)\cdot$$

Morph This is equivalent to *strong*. A morphism is strong if two different vertices in G map to different vertices in H.

$$x, y \in G \wedge x \sim y \Rightarrow \varphi(x) \sim \varphi(y)\cdot$$

Note that $x \neq y \Rightarrow \varphi(x) \neq \varphi(y)$ unless there is a loop in $G\cdot$

6 Other Functions

Here we keep a complete list of all of YAGS's functions not mentioned elsewhere.

1 ▶ `AddEdges(` G, E `)` O

Returns a new graph created from graph G by adding the edges in list E.

```
gap> g:=CycleGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
gap> AddEdges(g,[[1,3]]);
Graph( Category := SimpleGraphs, Order := 4, Size := 5, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 3 ] ] )
gap> AddEdges(g,[[1,3],[2,4]]);
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
```

–map

2 ▶ `Adjacencies(` G `)` O

Returns the adjacency lists of graph G.

```
gap> g:=PathGraph(3);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
gap> Adjacencies(g);
[ [ 2 ], [ 1, 3 ], [ 2 ] ]
```

–map

3 ▶ `Adjacency(` G, v `)` O

Returns the adjacency list of vertex v in G.

```
gap> g:=PathGraph(3);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
gap> Adjacency(g,1);
[ 2 ]
gap> Adjacency(g,2);
[ 1, 3 ]
```

–map

4 ▶ `AdjMatrix(` G `)` A

Returns the adjacency matrix of graph G.

```
gap> AdjMatrix(CycleGraph(4));
[ [ false, true, false, true ], [ true, false, true, false ],
  [ false, true, false, true ], [ true, false, true, false ] ]
```

–map

5 ► `AGraph` V

A 4-cycle with two pendant vertices on consecutive vertices of the cycle.

```
gap> AGraph;
Graph( Category := SimpleGraphs, Order := 6, Size := 6, Adjacencies :=
[ [ 2 ], [ 1, 3, 5 ], [ 2, 4 ], [ 3, 5 ], [ 2, 4, 6 ], [ 5 ] ] )
```

–map

6 ► `AntennaGraph` V

A `HouseGraph` with a pendant vertex (antenna) on the roof.

```
gap> AntennaGraph;
Graph( Category := SimpleGraphs, Order := 6, Size := 7, Adjacencies :=
[ [ 2, 4, 5 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3, 5 ], [ 1, 4, 6 ], [ 5 ] ] )
```

–map

FIXME AutomorphismGroup

7 ► `BackTrack(` *L*, *opts*, *chk*, *done*, *extra* `)` O

Generic, user-customizable backtracking algorithm.

A backtraking algorithm explores a decision tree in search for solutions to a combinatorial problem. The combinatorial problem and the search strategy are specified by the parameters:

L is just a list that `BackTrack` uses to keep track of solutions and partial solutions. It is usually set to the empty list as a starting point. After a solution is found, it is returned **and** stored in *L*. This value of *L* is then used as a starting point to search for the next solution in case `BackTrack` is called again. Partial solutions are also stored in *L* during the execution of `BackTrack`.

extra may be any object, list, record, etc. `BackTrack` only uses it to pass this data to the user-defined functions *opts*, *chk* and *done*, therefore offering you a way to share data between your functions.

opts`:=function(L,extra)` must return the list of continuation options (childs) one has after some partial solution (node) *L* has been reached within the decision tree (*opts* may use the extra data *extra* as needed). Each of the values in the list returned by *opts*`(L,extra)` will be tried as possible continuations of the partial solution *L*. If *opts*`(L,extra)` always returns the same list, you can put that list in place of the parameter *opts*.

chk`:=function(L,extra)` must evaluate the partial solution *L* possibly using the extra data *extra* and must return `false` when it knows that *L* can not be extended to a solution of the problem. Otherwise it returns `true`. *chk* may assume that *L*`[1..Length(L)-1]` already passed the test.

done`:=function(L,extra)` returns `true` if *L* is already a complete solution and `false` otherwise. In many combinatorial problems, any partial solution of certain length *N* is also a solution (and viceversa), so if this is your case, you can put that length in place of the parameter *done*.

The following example uses `BackTrack` in its simplest form to compute derrangements (permutations a set, where none of the elements appears in its original position).

```
gap> N:=4;;L:=[];;extra:=[];;opts:=[1..N];;done:=N;;
gap> chk:=function(L,extra) local i; i:=Length(L);
>           return not L[i] in L{[1..i-1]} and L[i]<> i; end;;
gap> BackTrack(L,opts,chk,done,extra);
[ 2, 1, 4, 3 ]
gap> BackTrack(L,opts,chk,done,extra);
[ 2, 3, 4, 1 ]
gap> BackTrack(L,opts,chk,done,extra);
[ 2, 4, 1, 3 ]
gap> BackTrack(L,opts,chk,done,extra);
[ 3, 1, 4, 2 ]
gap> BackTrack(L,opts,chk,done,extra);
[ 3, 4, 1, 2 ]
gap> BackTrack(L,opts,chk,done,extra);
[ 3, 4, 2, 1 ]
gap> BackTrack(L,opts,chk,done,extra);
[ 4, 1, 2, 3 ]
gap> BackTrack(L,opts,chk,done,extra);
[ 4, 3, 1, 2 ]
gap> BackTrack(L,opts,chk,done,extra);
[ 4, 3, 2, 1 ]
gap> BackTrack(L,opts,chk,done,extra);
fail
```

–map

8 ▶ `BackTrackBag(` *opts*, *chk*, *done*, *extra* `)` O

Returns the list of all solutions that would be returned one at a time by `Backtrack`.

The following example computes all derrangements of order 4.

```
gap> N:=4;;
gap> chk:=function(L,extra) local i; i:=Length(L);
>           return not L[i] in L{[1..i-1]} and L[i]<> i; end;;
gap> BackTrackBag([1..N],chk,N,[]);
[ [ 2, 1, 4, 3 ], [ 2, 3, 4, 1 ], [ 2, 4, 1, 3 ], [ 3, 1, 4, 2 ],
  [ 3, 4, 1, 2 ], [ 3, 4, 2, 1 ], [ 4, 1, 2, 3 ], [ 4, 3, 1, 2 ],
  [ 4, 3, 2, 1 ] ]
```

–map

9 ▶ `Basement(` G, KnG, x `)` O
▶ `Basement(` G, KnG, V `)` O

Given a graph G, some iterated clique graph KnG of G and a vertex x of KnG, the operation computes the *basement* of x with respect to G [Piz04]. Loosely speaking, the basement of x is the set of vertices of G that constitutes the iterated clique x.

```
gap> g:=Icosahedron;;Cliques(g);
[ [ 1, 2, 3 ], [ 1, 2, 6 ], [ 1, 3, 4 ], [ 1, 4, 5 ], [ 1, 5, 6 ],
  [ 4, 5, 7 ], [ 4, 7, 11 ], [ 5, 7, 8 ], [ 7, 8, 12 ], [ 7, 11, 12 ],
  [ 5, 6, 8 ], [ 6, 8, 9 ], [ 8, 9, 12 ], [ 2, 6, 9 ], [ 2, 9, 10 ],
  [ 9, 10, 12 ], [ 2, 3, 10 ], [ 3, 10, 11 ], [ 10, 11, 12 ], [ 3, 4, 11 ] ]
gap> kg:=CliqueGraph(g);; k2g:=CliqueGraph(kg);;
gap> Basement(g,k2g,1);Basement(g,k2g,2);
```

```
[ 1, 2, 3, 4, 5, 6 ]
[ 1, 2, 3, 4, 6, 10 ]
```

In its second form, V is a set of vertices of KnG, in that case, the basement is simply the union of the basements of the vertices in V.

```
gap> Basement(g,k2g,[1,2]);
[ 1, 2, 3, 4, 5, 6, 10 ]
```

–map

10 ► `BoxProduct(` G, H `)` O

Returns the box product, $G \square H$, of two graphs G and H (also known as the cartesian product).

The box product is calculated as follows:

For each pair of vertices $g \in G, h \in H$ we create a vertex (g, h). Given two such vertices (g, h) and (g', h') they are adjacent *iff* $g = g'$ and $h \sim h'$ or $g \sim g'$ and $h = h'$.

```
gap> g1:=PathGraph(3);g2:=CycleGraph(4);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
gap> g1g2:=BoxProduct(g1,g2);
Graph( Category := SimpleGraphs, Order := 12, Size := 20, Adjacencies :=
[ [ 2, 4, 5 ], [ 1, 3, 6 ], [ 2, 4, 7 ], [ 1, 3, 8 ], [ 1, 6, 8, 9 ],
  [ 2, 5, 7, 10 ], [ 3, 6, 8, 11 ], [ 4, 5, 7, 12 ], [ 5, 10, 12 ],
  [ 6, 9, 11 ], [ 7, 10, 12 ], [ 8, 9, 11 ] ] )
gap> VertexNames(g1g2);
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 1 ], [ 2, 2 ], [ 2, 3 ],
  [ 2, 4 ], [ 3, 1 ], [ 3, 2 ], [ 3, 3 ], [ 3, 4 ] ]
```

–map

11 ► `BoxTimesProduct(` G, H `)` O

Returns the boxtimes product of two graphs G and H, $G \boxtimes H$ (also known as the strong product).

The box times product is calculated as follows:

For each pair of vertices $g \in G, h \in H$ we create a vertex (g, h). Given two such vertices (g, h) and (g', h') such that $(g, h) \neq (g', h')$ they are adjacent *iff* $g \simeq g'$ and $h \simeq h'$.

```
gap> g1:=PathGraph(3);g2:=CycleGraph(4);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
gap> g1g2:=BoxTimesProduct(g1,g2);
Graph( Category := SimpleGraphs, Order := 12, Size := 36, Adjacencies :=
[ [ 2, 4, 5, 6, 8 ], [ 1, 3, 5, 6, 7 ], [ 2, 4, 6, 7, 8 ], [ 1, 3, 5, 7, 8 ],
  [ 1, 2, 4, 6, 8, 9, 10, 12 ], [ 1, 2, 3, 5, 7, 9, 10, 11 ],
  [ 2, 3, 4, 6, 8, 10, 11, 12 ], [ 1, 3, 4, 5, 7, 9, 11, 12 ],
  [ 5, 6, 8, 10, 12 ], [ 5, 6, 7, 9, 11 ], [ 6, 7, 8, 10, 12 ],
  [ 5, 7, 8, 9, 11 ] ] )
gap> VertexNames(g1g2);
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 1 ], [ 2, 2 ], [ 2, 3 ],
  [ 2, 4 ], [ 3, 1 ], [ 3, 2 ], [ 3, 3 ], [ 3, 4 ] ]
```

–map

12 ▶ `BullGraph` V

A triangle with two pendant vertices (horns).

```
gap> BullGraph;
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 2 ], [ 1, 3, 4 ], [ 2, 4 ], [ 2, 3, 5 ], [ 4 ] ] )
```

–map

13 ▶ `CayleyGraph(` *Grp*, *elms* `)` O
▶ `CayleyGraph(` *Grp* `)` O

Returns the graph G whose vertices are the elements of the group *Grp* such that x is adjacent to y iff $x * g = y$ for some g in the list *elms*. if *elms* is not provided, then the generators of G are used instead.

```
gap> grp:=Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> CayleyGraph(grp);
Graph( Category := SimpleGraphs, Order := 6, Size := 9, Adjacencies :=
[ [ 3, 4, 5 ], [ 3, 5, 6 ], [ 1, 2, 6 ], [ 1, 5, 6 ], [ 1, 2, 4 ],
  [ 2, 3, 4 ] ] )
gap> CayleyGraph(grp,[(1,2),(2,3)]);
Graph( Category := SimpleGraphs, Order := 6, Size := 6, Adjacencies :=
[ [ 2, 3 ], [ 1, 5 ], [ 1, 4 ], [ 3, 6 ], [ 2, 6 ], [ 4, 5 ] ] )
```

–map

14 ▶ `ChairGraph` V

A tree with degree sequence 3,2,1,1,1.

```
gap> ChairGraph;
Graph( Category := SimpleGraphs, Order := 5, Size := 4, Adjacencies :=
[ [ 2 ], [ 1, 3, 4 ], [ 2 ], [ 2, 5 ], [ 4 ] ] )
```

–map

15 ▶ `Circulant(` *n*, *jumps* `)` O

Returns the graph G whose vertices are [1..n] such that x is adjacent to y iff x+z=y mod n for some z the list of *jumps*

```
gap> Circulant(6,[1,2]);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 2, 3, 5, 6 ], [ 1, 3, 4, 6 ], [ 1, 2, 4, 5 ], [ 2, 3, 5, 6 ],
  [ 1, 3, 4, 6 ], [ 1, 2, 4, 5 ] ] )
```

–map

16 ▶ `ClawGraph` V

The graph on 4 vertices, 3 edges, and maximum degree 3.

```
gap> ClawGraph;
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2, 3, 4 ], [ 1 ], [ 1 ], [ 1 ] ] )
```

–map

17 ► `CliqueGraph( G )` A
► `CliqueGraph( G, m )` O

Returns the intersection graph of all the (maximal) cliques of G.

The additional parameter m aborts the computation when m cliques are found, even if they are all the cliques of G. If the bound m is reached, *fail* is returned.

```
gap> CliqueGraph(Octahedron);
Graph( Category := SimpleGraphs, Order := 8, Size := 24, Adjacencies :=
[ [ 2, 3, 4, 5, 6, 7 ], [ 1, 3, 4, 5, 6, 8 ], [ 1, 2, 4, 5, 7, 8 ],
  [ 1, 2, 3, 6, 7, 8 ], [ 1, 2, 3, 6, 7, 8 ], [ 1, 2, 4, 5, 7, 8 ],
  [ 1, 3, 4, 5, 6, 8 ], [ 2, 3, 4, 5, 6, 7 ] ] )
gap> CliqueGraph(Octahedron,9);
Graph( Category := SimpleGraphs, Order := 8, Size := 24, Adjacencies :=
[ [ 2, 3, 4, 5, 6, 7 ], [ 1, 3, 4, 5, 6, 8 ], [ 1, 2, 4, 5, 7, 8 ],
  [ 1, 2, 3, 6, 7, 8 ], [ 1, 2, 3, 6, 7, 8 ], [ 1, 2, 4, 5, 7, 8 ],
  [ 1, 3, 4, 5, 6, 8 ], [ 2, 3, 4, 5, 6, 7 ] ] )
gap> CliqueGraph(Octahedron,8);
fail
```

–map

18 ► `CliqueNumber( G )` A

Returns the order, $\omega(G)$, of a maximum clique of G.

```
gap> g:=SunGraph(4);
Graph( Category := SimpleGraphs, Order := 8, Size := 14, Adjacencies :=
[ [ 2, 8 ], [ 1, 3, 4, 6, 8 ], [ 2, 4 ], [ 2, 3, 5, 6, 8 ], [ 4, 6 ],
  [ 2, 4, 5, 7, 8 ], [ 6, 8 ], [ 1, 2, 4, 6, 7 ] ] )
gap> CliqueNumber(g);
4
```

–map

19 ► `Cliques( G )` A
► `Cliques( G, m )` O

Returns the set of all (maximal) cliques of a graph G. A clique is a maximal complete subgraph. Here, we use the Bron-Kerbosch algorithm [BK73].

In the second form, It stops computing cliques after m of them have been found.

```
gap> Cliques(Octahedron);
[ [ 1, 3, 5 ], [ 1, 3, 6 ], [ 1, 4, 5 ], [ 1, 4, 6 ], [ 2, 3, 5 ],
  [ 2, 3, 6 ], [ 2, 4, 5 ], [ 2, 4, 6 ] ]
gap> Cliques(Octahedron,4);
[ [ 1, 3, 5 ], [ 1, 3, 6 ], [ 1, 4, 5 ], [ 1, 4, 6 ] ]
```

–map

20 ► `ComplementGraph( G )` A

Computes the complement of graph G. The complement of a graph is created as follows: Create a graph G' with same vertices of G. For each x, $y \in G$ if $x \nsim y$ in G then $x \sim y$ in G'

```
gap> g:=ClawGraph;
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2, 3, 4 ], [ 1 ], [ 1 ], [ 1 ] ] )
gap> ComplementGraph(g);
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [  ], [ 3, 4 ], [ 2, 4 ], [ 2, 3 ] ] )
```

–map

21 ▶ `CompleteBipartiteGraph(` *n*`,` *m* `)` F

Returns the complete bipartite whose parts have order n and m respectively. This is the joint (Zykov sum) of two discrete graphs of order n and m.

```
gap> CompleteBipartiteGraph(2,3);
Graph( Category := SimpleGraphs, Order := 5, Size := 6, Adjacencies :=
[ [ 3, 4, 5 ], [ 3, 4, 5 ], [ 1, 2 ], [ 1, 2 ], [ 1, 2 ] ] )
```

–map

22 ▶ `CompleteGraph(` *n* `)` F

Returns the complete graph of order n. A complete graph is a graph where all vertices are connected to each other.

```
gap> CompleteGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
```

–map

23 ▶ `CompleteMultipartiteGraph(` *n1*`,` *n2* `[,` *n3* `...] )` F

Returns the complete multipartite graph where the orders of the parts are *n1*, *n2*, ... It is also the Zykov sum of discrete graphs of order *n1*, *n2*, ...

```
gap> CompleteMultipartiteGraph(2,2,2);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 3, 4, 5, 6 ], [ 3, 4, 5, 6 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ],
  [ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ] ] )
```

–map

24 ▶ `CompletesOfGivenOrder(` *G*`,` *o* `)` O

This operation finds all complete subgraphs of order o in graph G.

```
gap> g:=SunGraph(4);
Graph( Category := SimpleGraphs, Order := 8, Size := 14, Adjacencies :=
[ [ 2, 8 ], [ 1, 3, 4, 6, 8 ], [ 2, 4 ], [ 2, 3, 5, 6, 8 ], [ 4, 6 ],
  [ 2, 4, 5, 7, 8 ], [ 6, 8 ], [ 1, 2, 4, 6, 7 ] ] )
gap> CompletesOfGivenOrder(g,3);
[ [ 1, 2, 8 ], [ 2, 3, 4 ], [ 2, 4, 6 ], [ 2, 4, 8 ], [ 2, 6, 8 ],
  [ 4, 5, 6 ], [ 4, 6, 8 ], [ 6, 7, 8 ] ]
gap> CompletesOfGivenOrder(g,4);
[ [ 2, 4, 6, 8 ] ]
```

–map

25 ▶ `Composition( G, H )` O

Returns the composition $G[H]$ of two graphs G and H.

A composition of graphs is obtained by calculating the GraphSum of G with *Order(G)* copies of H, $G[H] = GraphSum(G, [H, \dots, H])$.

```
gap> g1:=CycleGraph(4);;g2:=DiscreteGraph(2);;
gap> Composition(g1,g2);
Graph( Category := SimpleGraphs, Order := 8, Size := 16, Adjacencies :=
[ [ 3, 4, 7, 8 ], [ 3, 4, 7, 8 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ],
  [ 3, 4, 7, 8 ], [ 3, 4, 7, 8 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ] ] )
```

–map

26 ▶ `Coordinates( G )` O

Gets the coordinates of the vertices of G, which are used to draw G by `Draw( G )`. If the coordinates have not been previously set, `Coordinates` returns *fail*.

```
gap> g:=CycleGraph(4);;
gap> Coordinates(g);
fail
gap> SetCoordinates(g,[[-10,-10 ],[-10,20],[20,-10 ], [20,20]]);
gap> Coordinates(g);
[ [ -10, -10 ], [ -10, 20 ], [ 20, -10 ], [ 20, 20 ] ]
```

–map

27 ▶ `CopyGraph( G )` O

Returns a fresh copy of graph G. Only the order and adjacency information is copied, all other known attributes of G are not. Mainly used to transform a graph from one category to another. The new graph will be forced to comply with the `TargetGraphCategory`.

```
gap> g:=CompleteGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
gap> g1:=CopyGraph(g:GraphCategory:=OrientedGraphs);
Graph( Category := OrientedGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 3, 4 ], [ 4 ], [  ] ] )
gap> CopyGraph(g1:GraphCategory:=SimpleGraphs);
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
```

–map

28 ▶ `CuadraticRingGraph( Rng )` O

Returns the graph G whose vertices are the elements of *Rng* such that x is adjacent to y iff x+zˆ2=y for some z in *Rng*

```
gap> CuadraticRingGraph(ZmodnZ(8));
Graph( Category := SimpleGraphs, Order := 8, Size := 12, Adjacencies :=
[ [ 2, 5, 8 ], [ 1, 3, 6 ], [ 2, 4, 7 ], [ 3, 5, 8 ], [ 1, 4, 6 ],
  [ 2, 5, 7 ], [ 3, 6, 8 ], [ 1, 4, 7 ] ] )
```

–map

29 ► `Cube` V

The 1-skeleton of Plato's cube.

```
gap> Cube;
Graph( Category := SimpleGraphs, Order := 8, Size := 12, Adjacencies :=
[ [ 2, 3, 5 ], [ 1, 4, 6 ], [ 1, 4, 7 ], [ 2, 3, 8 ], [ 1, 6, 7 ],
 [ 2, 5, 8 ], [ 3, 5, 8 ], [ 4, 6, 7 ] ] )
```

–map

30 ► `CubeGraph(` *n* `)` F

Returns the hypercube of dimension n. This is the box product (cartesian product) of n copies of K_2 (an edge).

```
gap> CubeGraph(3);
Graph( Category := SimpleGraphs, Order := 8, Size := 12, Adjacencies :=
[ [ 2, 3, 5 ], [ 1, 4, 6 ], [ 1, 4, 7 ], [ 2, 3, 8 ], [ 1, 6, 7 ],
[ 2, 5, 8 ], [ 3, 5, 8 ], [ 4, 6, 7 ] ] )
```

-map

31 ► `CycleGraph(` *n* `)` F

Returns the cyclic graph on n vertices.

```
gap> CycleGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 2, 5 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 1, 4 ] ] )
```

–map

32 ► `CylinderGraph(` *Base*, *Height* `)` F

Returns a cylinder of base *Base* and height *Height*. The order of this graph is *Base*(Height+1)* and it is constructed by taking *Height+1* copies of the cyclic graph on *Base* vertices, ordering these cycles linearly and then joining consecutive cycles by a zigzagging *2*Base*-cycle. This graph is a triangulation of the cylinder where all internal vertices are of degree 6 and the border vertices are of degree 4.

```
gap> g:=CylinderGraph(4,1);
Graph( Category := SimpleGraphs, Order := 8, Size := 16, Adjacencies :=
[ [ 2, 4, 5, 6 ], [ 1, 3, 6, 7 ], [ 2, 4, 7, 8 ], [ 1, 3, 5, 8 ],
  [ 1, 4, 6, 8 ], [ 1, 2, 5, 7 ], [ 2, 3, 6, 8 ], [ 3, 4, 5, 7 ] ] )
gap> g:=CylinderGraph(4,2);
Graph( Category := SimpleGraphs, Order := 12, Size := 28, Adjacencies :=
[ [ 2, 4, 5, 6 ], [ 1, 3, 6, 7 ], [ 2, 4, 7, 8 ], [ 1, 3, 5, 8 ],
  [ 1, 4, 6, 8, 9, 10 ], [ 1, 2, 5, 7, 10, 11 ], [ 2, 3, 6, 8, 11, 12 ],
  [ 3, 4, 5, 7, 9, 12 ], [ 5, 8, 10, 12 ], [ 5, 6, 9, 11 ], [ 6, 7, 10, 12 ],
  [ 7, 8, 9, 11 ] ] )
```

–map

33 ► `DartGraph` V

A diamond with a pending vertex and maximum degree 4.

```
gap> DartGraph;
Graph( Category := SimpleGraphs, Order := 5, Size := 6, Adjacencies :=
[ [ 2 ], [ 1, 3, 4, 5 ], [ 2, 4, 5 ], [ 2, 3 ], [ 2, 3 ] ] )
```

–map

34 ▶ `DeclareQtfyProperty(` *Name*, *Filter* `)` F

For internal use.

Declares a YAGS quantifiable property named *Name* for filter *Filter*. This in turns, declares a boolean GAP property *Name* and an integer GAP attribute *QtfyName*.

The user must provide the method *Name*(*O*, *qtfy*). If *qtfy* is false, the method must return a boolean indicating whether the property holds, otherwise, the method must return a non-negative integer quantifying how far is the object from satisfying the property. In the latter case, returning 0 actually means that the object does satisfy the property.

```
gap> DeclareQtfyProperty("Is2Regular",Graphs);
gap> InstallMethod(Is2Regular,"for graphs",true,[Graphs,IsBool],0,
> function(G,qtfy)
>   local x,count;
>   count:=0;
>   for x in Vertices(G) do
>     if VertexDegree(G,x)<> 2 then
>       if not qtfy then
>         return false;
>       fi;
>         count:=count+1;
>     fi;
>   od;
>   if not qtfy then return true; fi;
>   return count;
> end);
gap> Is2Regular(CycleGraph(4));
true
gap> QtfyIs2Regular(CycleGraph(4));
0
gap> Is2Regular(DiamondGraph);
false
gap> QtfyIs2Regular(DiamondGraph);
2
```

–map

35 ▶ `Diameter(` *G* `)` A

Returns the maximum among the distances between pairs of vertices of *G*.

```
gap> g:=CycleGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 2, 5 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 1, 4 ] ] )
gap> Diameter(g);
2
```

–map

36 ► `DiamondGraph` V

The graph on 4 vertices and 5 edges.

```
gap> DiamondGraph;
Graph( Category := SimpleGraphs, Order := 4, Size := 5, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 3 ] ] )
```

–map

37 ► `DiscreteGraph(` n `)` F

Returns the discrete graph of order n. A discrete graph is a graph without edges.

```
gap> DiscreteGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 0, Adjacencies :=
[ [  ], [  ], [  ], [  ] ] )
```

–map

38 ► `DisjointUnion(` G, H `)` O

Returns the disjoint union of two graphs G and H, $G \dot\cup H$.

```
gap> g1:=PathGraph(3);g2:=PathGraph(2);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies :=
[ [ 2 ], [ 1 ] ] )
gap> DisjointUnion(g1,g2);
Graph( Category := SimpleGraphs, Order := 5, Size := 3, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ], [ 5 ], [ 4 ] ] )
```

–map

39 ► `Distance(` G, x, y `)` O

Returns the length of a minimal path connecting x to y in G.

```
gap> Distance(CycleGraph(5),1,3);
2
gap> Distance(CycleGraph(5),1,5);
1
```

–map

40 ► `DistanceGraph(` G, D `)` O

Given a graph G and list of distances D, `DistanceGraph` returns the new graph constructed on the vertices of G where two vertices are adjacent iff the distance (in G) between them belongs to the list D.

```
gap> g:=CycleGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 2, 5 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 1, 4 ] ] )
gap> DistanceGraph(g,[2]);
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 3, 4 ], [ 4, 5 ], [ 1, 5 ], [ 1, 2 ], [ 2, 3 ] ] )
gap> DistanceGraph(g,[1,2]);
Graph( Category := SimpleGraphs, Order := 5, Size := 10, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 4, 5 ], [ 1, 2, 4, 5 ], [ 1, 2, 3, 5 ],
  [ 1, 2, 3, 4 ] ] )
```

–map

41 ► `DistanceMatrix( G )` A

Returns the distance matrix D of a graph G: D[x][y] is the distance in G from vertex x to vertex y. The matrix may be asymmetric if the graph is not simple. An infinite entry in the matrix means that there is no path between the vertices. Floyd's algorithm is used to compute the matrix.

```
gap> g:=PathGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3 ] ] )
gap> Display(DistanceMatrix(g));
[ [  0,  1,  2,  3 ],
  [  1,  0,  1,  2 ],
  [  2,  1,  0,  1 ],
  [  3,  2,  1,  0 ] ]
gap> g:=PathGraph(4:GraphCategory:=OrientedGraphs);
Graph( Category := OrientedGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2 ], [ 3 ], [ 4 ], [  ] ] )
gap> Display(DistanceMatrix(g));
[ [         0,         1,         2,         3 ],
  [  infinity,         0,         1,         2 ],
  [  infinity,  infinity,         0,         1 ],
  [  infinity,  infinity,  infinity,         0 ] ]
```

–map

42 ► `Distances( G, A, B )` O

Given two lists of vertices A, B of a graph G, `Distances` returns the list of distances for every pair in the cartesian product of A and B. The order of the vertices in lists A and B affects the order of the list of distances returned.

```
gap> g:=CycleGraph(5);;
gap> Distances(g, [1,3], [2,4]);
[ 1, 2, 1, 1 ]
gap> Distances(g, [3,1], [2,4]);
[ 1, 1, 1, 2 ]
```

–map

43 ► `DistanceSet( G, A, B )` O

Given two subsets of vertices A, B of a graph G, `DistanceSet` returns the set of distances for every pair in the cartesian product of A and B.

```
gap> g:=CycleGraph(5);;
gap> DistanceSet(g, [1,3], [2,4]);
[ 1, 2 ]
```

–map

44 ► `Dodecahedron` V

The 1-skeleton of Plato's Dodecahedron.

gap¿ Dodecahedron; Graph(Category := SimpleGraphs, Order := 20, Size := 30, Adjacencies := [[2, 5, 6], [1, 3, 7], [2, 4, 8], [3, 5, 9], [1, 4, 10], [1, 11, 15], [2, 11, 12], [3, 12, 13], [4, 13, 14], [5, 14, 15], [6, 7, 16], [7, 8, 17], [8, 9, 18], [9, 10, 19], [6, 10, 20], [11, 17, 20], [12, 16, 18], [13, 17, 19], [14, 18, 20], [15, 16, 19]])

–map

45 ► `DominoGraph` V

Two squares glued by an edge.

```
gap> DominoGraph;
Graph( Category := SimpleGraphs, Order := 6, Size := 7, Adjacencies :=
[ [ 2, 4, 6 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3, 5 ], [ 4, 6 ], [ 1, 5 ] ] )
```

–map

46 ► `Draw(` G `)` O

Takes a graph G and makes a drawing of it in a separate window. The user can then view and modify the drawing and finaly save the vertex coordinates of the drawing into the graph G.

Within the separate window, type h to toggle on/off the help menu. Besides the keyword commands indicated in the help menu, the user may also move vertices (by dragging them), move the whole drawing (by dragging the background) and scale the drawing (by using the mouse wheel).

```
gap> Coordinates(Icosahedron);
fail
gap> Draw(Icosahedron);
gap> Coordinates(Icosahedron);
[ [ 29, -107 ], [ 65, -239 ], [ 240, -62 ], [ 78, 79 ], [ -107, 28 ],
  [ -174, -176 ], [ -65, 239 ], [ -239, 62 ], [ -78, -79 ], [ 107, -28 ],
  [ 174, 176 ], [ -29, 107 ] ]
```

This preliminary version, should work fine on GNU/Linux. For other plataforms, you should probably (at least) set up correctly the variable `drawproc` which should point to the correct external program binary. Java binaries are provided for GNU/Linux, Mac OS X and Windows.

```
gap> drawproc;
"/usr/share/gap/pkg/yags/bin/draw/application.linux64/draw"
```

–map

47 ► `DumpObject(` O `)` O

Dumps all information available for object O. This information includes to which categories it belongs as well as its type and hashing information used by GAP.

```
gap> DumpObject( true );
Object( TypeObj := NewType( NewFamily( "BooleanFamily", [ 11 ], [ 11 ] ),
[ 11, 34 ] ), Categories := [ "IS_BOOL" ] )
```

–map

48 ► `Edges(` G `)` O

Returns the list of edges of graph G.

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

–map

49 ► `Excentricity(` G, x `)` F

Returns the distance from a vertex x in graph G to its most distant vertex in G.

```
gap> g:=PathGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 4, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4 ] ] )
gap> Excentricity(g,1);
4
gap> Excentricity(g,3);
2
```

–map

50 ▶ `FanGraph(` *N* `)` F

Returns the *N*-Fan: The join of a vertex and a *(N+1)*-path.

```
gap> FanGraph(4);
Graph( Category := SimpleGraphs, Order := 6, Size := 9, Adjacencies :=
[ [ 2, 3, 4, 5, 6 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 3, 5 ], [ 1, 4, 6 ],
[ 1, 5 ] ] )
```

–map

51 ▶ `FishGraph` V

A square and a triangle glued by a vertex.

```
gap> FishGraph;
Graph( Category := SimpleGraphs, Order := 6, Size := 7, Adjacencies :=
[ [ 2, 3, 4, 6 ], [ 1, 3 ], [ 1, 2 ], [ 1, 5 ], [ 4, 6 ], [ 1, 5 ] ] )
```

–map

52 ▶ `GemGraph` V

The 3-Fan graph.

```
gap> GemGraph;
Graph( Category := SimpleGraphs, Order := 5, Size := 7, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 3, 5 ], [ 1, 4 ] ] )
```

–map

53 ▶ `Graph(` *R* `)` O

Returns a new graph created from the record *R*. The record must provide the field *Category* and either the field *Adjacencies* or the field *AdjMatrix*

```
gap> Graph(rec(Category:=SimpleGraphs,Adjacencies:=[[2],[1]]));
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies := [ [ 2 ], [ 1 ] ] )
gap> Graph(rec(Category:=SimpleGraphs,AdjMatrix:=[[false, true],[true, false]]));
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies := [ [ 2 ], [ 1 ] ] )
```

Its main purpose is to import graphs from files, which could have been previously exported using `PrintTo`.

```
gap> g:=CycleGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
gap> PrintTo("aux.g","h1:=",g,";");
gap> Read("aux.g");
gap> h1;
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
```

–map

54 ► `GraphByAdjacencies(` A `)` F

Returns a new graph having A as its list of adjacencies. The order of the created graph is `Length(A)`, and the set of neighbors of vertex x is $A[x]$.

```
gap> GraphByAdjacencies([[2],[1,3],[2]]);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
```

Note, however, that the graph is forced to comply with the `TargetGraphCategory`.

```
gap> GraphByAdjacencies([[1,2,3],[],[]]);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2, 3 ], [ 1 ], [ 1 ] ] )
```

–map

55 ► `GraphByAdjMatrix(` M `)` F

Returns a new graph created from an adjacency matrix M. The matrix M must be a square boolean matrix.

```
gap> m:=[ [ false, true, false ], [ true, false, true ], [ false, true, false ] ];;
gap> g:=GraphByAdjMatrix(m);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
gap> AdjMatrix(g);
[ [ false, true, false ], [ true, false, true ], [ false, true, false ] ]
```

Note, however, that the graph is forced to comply with the `TargetGraphCategory`.

```
gap> m:=[ [ true, true], [ false, false ] ];;
gap> g:=GraphByAdjMatrix(m);
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies := [ [ 2 ], [ 1 ] ] )
gap> AdjMatrix(g);
[ [ false, true ], [ true, false ] ]
```

–map

56 ► `GraphByCompleteCover(` C `)` F

Returns the minimal graph where the elements of C are (the vertex sets of) complete subgraphs.

```
gap> GraphByCompleteCover([[1,2,3,4],[4,6,7]]);
Graph( Category := SimpleGraphs, Order := 7, Size := 9, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3, 6, 7 ], [  ], [ 4, 7 ],
  [ 4, 6 ] ] )
```

–map

57 ► `GraphByRelation(` V `,` R `)` F
► `GraphByRelation(` N `,` R `)` F

Returns a new graph created from a set of vertices V and a binary relation R, where $x \sim y$ iff $R(x,y) = true$. In the second form, N is an integer and V is assumed to be $\{1, 2, \ldots, N\}$.

```
gap> R:=function(x,y) return Intersection(x,y)<>[]; end;;
gap> GraphByRelation([[1,2,3],[3,4,5],[5,6,7]],R);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
gap> GraphByRelation(8,function(x,y) return AbsInt(x-y)<=2; end);
Graph( Category := SimpleGraphs, Order := 8, Size := 13, Adjacencies :=
[ [ 2, 3 ], [ 1, 3, 4 ], [ 1, 2, 4, 5 ], [ 2, 3, 5, 6 ], [ 3, 4, 6, 7 ],
  [ 4, 5, 7, 8 ], [ 5, 6, 8 ], [ 6, 7 ] ] )
```

–map

58 ► `GraphByWalks(` *walk1* `,` *walk2* `, ... )` F

Returns the minimal graph such that *walk1*, *walk2*, etc are walks.

```
gap> GraphByWalks([1,2,3,4,1],[1,5,6]);
Graph( Category := SimpleGraphs, Order := 6, Size := 6, Adjacencies :=
[ [ 2, 4, 5 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ], [ 1, 6 ], [ 5 ] ] )
```

Walks can be *nested*, which greatly improves the versatility of this function.

```
gap> GraphByWalks([1,[2,3,4],5],[5,6]);
Graph( Category := SimpleGraphs, Order := 6, Size := 9, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 5 ], [ 1, 2, 4, 5 ], [ 1, 3, 5 ], [ 2, 3, 4, 6 ], [ 5 ] ] )
```

–map

59 ► `GraphCategory( [`G`, ... ] )` F

Returns the minimal common category to a list of graphs. See Section 2 for the relationship among categories.

If the list is empty the default category is returned.

60 ► `Graphs( )` C

Graphs are the base category used by YAGS. This category contains all graphs that can be represented in YAGS.

61 ► `GraphSum(` G `,` L `)` O

Returns the lexicographic sum of a list of graphs L over a graph G.

The lexicographic sum is computed as follows:

Given G, with $Order(G) = n$ and a list of n graphs $L = [G_1, \ldots, G_n]$, We take the disjoint union of $G_1, G_2, \ldots, G_n$ and then we add all the edges between G_i and G_j whenever $[i,j]$ is and edge of G.

If L contains holes, the trivial graph is used in place.

```
gap> t:=TrivialGraph;; g:=CycleGraph(4);;
gap> GraphSum(PathGraph(3),[t,g,t]);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 5, 6 ], [ 1, 2, 4, 6 ], [ 1, 3, 5, 6 ],
  [ 1, 2, 4, 6 ], [ 2, 3, 4, 5 ] ] )
gap> GraphSum(PathGraph(3),[,g,]);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 5, 6 ], [ 1, 2, 4, 6 ], [ 1, 3, 5, 6 ],
  [ 1, 2, 4, 6 ], [ 2, 3, 4, 5 ] ] )
```

–map

62 ► `GraphToRaw(` *filename*, *G* `)` O

Converts a YAGS graph G into a raw format (number of vertices, coordinates and adjacency matrix) and writes the converted data to the file *filename*. For use by the external program `draw` (see `Draw(`G`)`).

```
gap> g:=CycleGraph(4);;
gap> GraphToRaw("mygraph.raw",g);
```

–map

63 ► `GraphUpdateFromRaw(` *filename*, *G* `)` O

Updates the coordinates of G from a file *filename* in raw format. Intended for internal use only.

–map

64 ► `GroupGraph(` *G*, *Grp*, *act* `)` O
► `GroupGraph(` *G*, *Grp* `)` O

Given a graph G, a group *Grp* and an action *act* of *Grp* in some set S which contains Vertices(G), GroupGraph returns a new graph with vertex set $\{act(v,g) : g \in Grp, v \in Vertices(G)\}$ and edge set $\{\{act(v,g), act(u,g)\} : g\ inGrp\{u,v\} \in Edges(G)\}$.

If *act* is omited, the standard GAP action `OnPoints` is used.

```
gap> GroupGraph(GraphByWalks([1,2]),Group([(1,2,3,4,5),(2,5)(3,4)]));
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 2, 5 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 1, 4 ] ] )
```

–map

65 ► `HouseGraph` V

A 4-Cycle and a triangle glued by an edge.

```
gap> HouseGraph;
Graph( Category := SimpleGraphs, Order := 5, Size := 6, Adjacencies :=
[ [ 2, 4, 5 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3, 5 ], [ 1, 4 ] ] )
```

–map

66 ► `Icosahedron` V

The 1-skeleton of Plato's icosahedron.

```
gap> Icosahedron;
Graph( Category := SimpleGraphs, Order := 12, Size := 30, Adjacencies :=
[ [ 2, 3, 4, 5, 6 ], [ 1, 3, 6, 9, 10 ], [ 1, 2, 4, 10, 11 ],
  [ 1, 3, 5, 7, 11 ], [ 1, 4, 6, 7, 8 ], [ 1, 2, 5, 8, 9 ],
  [ 4, 5, 8, 11, 12 ], [ 5, 6, 7, 9, 12 ], [ 2, 6, 8, 10, 12 ],
  [ 2, 3, 9, 11, 12 ], [ 3, 4, 7, 10, 12 ], [ 7, 8, 9, 10, 11 ] ] )
```

–map

67 ▶ `in(` G, C `)` O

Returns `true` if graph G belongs to category C and `false` otherwise.

68 ▶ `InducedSubgraph(` G, V `)` O

Returns the subgraph of graph G induced by the vertex set V.

```
gap> g:=CycleGraph(6);
Graph( Category := SimpleGraphs, Order := 6, Size := 6, Adjacencies :=
[ [ 2, 6 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4, 6 ], [ 1, 5 ] ] )
gap> InducedSubgraph(g,[3,4,6]);
Graph( Category := SimpleGraphs, Order := 3, Size := 1, Adjacencies :=
[ [ 2 ], [ 1 ], [  ] ] )
```

The order of the elements in V does matter.

```
gap> InducedSubgraph(g,[6,3,4]);
Graph( Category := SimpleGraphs, Order := 3, Size := 1, Adjacencies :=
[ [  ], [ 3 ], [ 2 ] ] )
```

–map

69 ▶ `InNeigh(` G, x `)` O

Returns the list of in-neighbors of x in G.

```
gap> tt:=CompleteGraph(5:GraphCategory:=OrientedGraphs);
Graph( Category := OrientedGraphs, Order := 5, Size := 10, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 3, 4, 5 ], [ 4, 5 ], [ 5 ], [  ] ] )
gap> InNeigh(tt,3);
[ 1, 2 ]
gap> OutNeigh(tt,3);
[ 4, 5 ]
```

–map

70 ▶ `IntersectionGraph(` L `)` F

Returns the intersection graph of the family of sets L. This graph has a vertex for every set in L, and two such vertices are adjacent iff the corresponding sets have non-empty intersection.

```
gap> IntersectionGraph([[1,2,3],[3,4,5],[5,6,7]]);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
```

–map

71 ▶ `IsBoolean(` O `)` F

Returns `true` if object O is `true` or `false` and `false` otherwise.

```
gap> IsBoolean( true ); IsBoolean( fail ); IsBoolean ( false );
true
false
true
```

–map

72 ▶ `IsCliqueGated(` G `)` A

Returns `true` if G is a clique gated graph [HK96].

–map

73 ▶ `IsCliqueHelly(` G `)` A

Returns `true` if the set of (maximal) cliques G satisfy the *Helly* property.

The Helly property is defined as follows:

A non-empty family $\mathcal{F}$ of non-empty sets satisfies the Helly property if every pairwise intersecting subfamily of $\mathcal{F}$ has a non-empty total intersection.

Here we use the Dragan-Szwarcfiter characterization [Dra89,Szw97] to compute the Helly property.

```
gap> g:=SunGraph(3);
Graph( Category := SimpleGraphs, Order := 6, Size := 9, Adjacencies :=
[ [ 2, 6 ], [ 1, 3, 4, 6 ], [ 2, 4 ], [ 2, 3, 5, 6 ], [ 4, 6 ],
  [ 1, 2, 4, 5 ] ] )
gap> IsCliqueHelly(g);
false
```

–map

74 ▶ `IsComplete(` G, L `)` O

Returns true if L induces a complete subgraph of G.

```
gap> IsComplete(DiamondGraph,[1,2,3]);
true
gap> IsComplete(DiamondGraph,[1,2,4]);
false
```

–map

75 ▶ `IsCompleteGraph(` G `)` P
▶ `QtfyIsCompleteGraph(` G `)` A

Returns `true` if graph G is a complete graph, `false` otherwise. In a complete graph every pair of vertices is an edge.

–map

76 ▶ `IsDiamondFree(` G, *qtfy* `)` P

Returns `true` if G is free from induced diamonds, `false` otherwise.

```
gap> IsDiamondFree(Cube);
true
gap> IsDiamondFree(Octahedron);
false
```

–map

77 ▶ `IsEdge(` G `, [x, y] )` O

Returns true if [x,y] is an edge of G.

```
gap> IsEdge(PathGraph(3),[1,2]);
true
gap> IsEdge(PathGraph(3),[1,3]);
false
```

–map

78 ▶ `IsIsomorphicGraph(` G, H `)` O

Returns `true` when G is isomorphic to H and `false` otherwise.

```
gap> g:=PowerGraph(CycleGraph(6),2);;h:=Octahedron;;
gap> IsIsomorphicGraph(g,h);
true
```

–map

79 ▶ `IsLoopless(` G `)` P
▶ `QtfyIsLoopless(` G `)` A

Returns `true` if graph G have no loops, `false` otherwise. Loops are edges from a vertex to itself.

–map

80 ▶ `IsoMorphism(` G, H `)` O
▶ `NextIsoMorphism(` G, H, f `)` O

`IsoMorphism` returns one isomorphism from G to H. `NextIsoMorphism` returns the next isomorphism from G to H in the lexicographic order, it returns `fail` if there are no more isomorphisms. If G has n vertices, an isomorphisms $f : G \to H$ is represented as the list `[f(1), f(2), ..., f(n)]`.

```
gap> g:=CycleGraph(4);;h:=CompleteBipartiteGraph(2,2);;
gap> f:=IsoMorphism(g,h);
[ 1, 3, 2, 4 ]
gap> NextIsoMorphism(g,h,f);
[ 1, 4, 2, 3 ]
gap> NextIsoMorphism(g,h,f);
[ 2, 3, 1, 4 ]
gap> NextIsoMorphism(g,h,f);
[ 2, 4, 1, 3 ]
```

–map

81 ▶ `IsoMorphisms(` G, H `)` O

Returns the list of all isomorphism from G to H. If G has n vertices, an isomorphisms $f : G \to H$ is represented as the list `[f(1), f(2), ..., f(n)]`.

```
gap> g:=CycleGraph(4);;h:=CompleteBipartiteGraph(2,2);;
gap> IsoMorphisms(g,h);
[ [ 1, 3, 2, 4 ], [ 1, 4, 2, 3 ], [ 2, 3, 1, 4 ], [ 2, 4, 1, 3 ],
  [ 3, 1, 4, 2 ], [ 3, 2, 4, 1 ], [ 4, 1, 3, 2 ], [ 4, 2, 3, 1 ] ]
```

–map

82 ▶ `IsOriented( G )` P
▶ `QtfyIsOriented( G )` A

Returns `true` if graph G is an oriented graph, `false` otherwise. Regardless of the categories that G belongs to, G is oriented if whenever `[x,y]` is an edge of G, `[y,x]` is not.

–map

83 ▶ `IsSimple( G )` O

Returns `true` if graph G is a simple graph, `false` otherwise. Regardless of the categories that G belongs to, G is simple if and only if G is undirected and loopless.

Returns `true` if the graph G is simple regardless of its category.

–map

84 ▶ `IsTournament( G )` O

Returns `true` if G is a tournament.

```
gap> tt:=CompleteGraph(5:GraphCategory:=OrientedGraphs);
Graph( Category := OrientedGraphs, Order := 5, Size := 10, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 3, 4, 5 ], [ 4, 5 ], [ 5 ], [  ] ] )
gap> IsTournament(tt);
true
```

–map

85 ▶ `IsTransitiveTournament( G )` O

Returns `true` if G is a transitive tournament.

```
gap> tt:=CompleteGraph(5:GraphCategory:=OrientedGraphs);
Graph( Category := OrientedGraphs, Order := 5, Size := 10, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 3, 4, 5 ], [ 4, 5 ], [ 5 ], [  ] ] )
gap> IsTransitiveTournament(tt);
true
```

–map

86 ▶ `IsUndirected( G )` P
▶ `QtfyIsUndirected( G )` A

Returns `true` if graph G is an undirected graph, `false` otherwise. Regardless of the categories that G belongs to, G is undirected if whenever `[x,y]` is an edge of G, `[y,x]` is also an egde of G.

–map

87 ▶ `JohnsonGraph( n, r )` F

Returns the Johnson graph $J(n,r)\cdot$ A Johnson Graph is a graph constructed as follows. Each vertex represents a subset of the set $\{1,\ldots,n\}$ with cardinality $r\cdot$

$$V(J(n,r)) = \{X \subset \{1,\ldots,n\} || X| = r\}$$

and there is an edge between two vertices if and only if the cardinality of the intersection of the sets they represent is $r-1$

$$X \sim X' \text{ iff } |X \cup X'| = r - 1\cdot$$

```
gap> JohnsonGraph(4,2);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 4, 6 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ],
  [ 1, 3, 4, 6 ], [ 2, 3, 4, 5 ] ] )
```

–map

88 ► `Join(` G, H `)` O

Returns the result of joining graph G and H, $G + H$ (also known as the Zykov sum).

Joining graphs is computed as follows:

First, we obtain the disjoint union of graphs G and H. Second, for each vertex $g \in G$ we add an edge to each vertex $h \in H$.

```
gap> g1:=DiscreteGraph(2);g2:=CycleGraph(4);
Graph( Category := SimpleGraphs, Order := 2, Size := 0, Adjacencies :=
[ [  ], [  ] ] )
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
gap> Join(g1,g2);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 3, 4, 5, 6 ], [ 3, 4, 5, 6 ], [ 1, 2, 4, 6 ], [ 1, 2, 3, 5 ],
  [ 1, 2, 4, 6 ], [ 1, 2, 3, 5 ] ] )
```

–map

89 ► `KiteGraph` V

A diamond with a pending vertex and maximum degree 3.

```
gap> KiteGraph;
Graph( Category := SimpleGraphs, Order := 5, Size := 6, Adjacencies :=
[ [ 2 ], [ 1, 3, 4 ], [ 2, 4, 5 ], [ 2, 3, 5 ], [ 3, 4 ] ] )
```

–map

90 ► `LineGraph(` G `)` O

Returns the line graph *L(G)* of graph G. The line graph is the intersection graph of the edges of G, *i.e.* the vertices of $L(G)$ are the edges of G two of them being adjacent iff they are incident.

```
gap> g:=Tetrahedron;
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
gap> LineGraph(g);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 4, 6 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ],
  [ 1, 3, 4, 6 ], [ 2, 3, 4, 5 ] ] )
```

–map

91 ► `LooplessGraphs( )` C

Loopless Graphs are graphs which have no loops.

92 ► `MaxDegree(` G `)` O

Returns the maximum degree in graph G.

```
gap> g:=GemGraph;
Graph( Category := SimpleGraphs, Order := 5, Size := 7, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 3, 5 ], [ 1, 4 ] ] )
gap> MaxDegree(g);
4
```

–map

93 ▶ `MinDegree(` G `)` O

Returns the minimum degree in graph G.

```
gap> g:=GemGraph;
Graph( Category := SimpleGraphs, Order := 5, Size := 7, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 3, 5 ], [ 1, 4 ] ] )
gap> MinDegree(g);
2
```

–map

FIXME NextIsoMorphism

94 ▶ `NextPropertyMorphism(` $G1$, $G2$, m, c `)` O

Returns the next morphisms that is true for the list of checks c given graphs $G1$ and $G2$ starting with (possibly incomplete) morphism m. Note that if m is a variable the operation will change its value to the result of the operation.

```
gap> f:=[];;
gap> NextPropertyMorphism(CycleGraph(4),CompleteGraph(4),f,[CHQ_MONO,CHQ_MORPH$
[ 1, 2, 3, 4 ]
gap> NextPropertyMorphism(CycleGraph(4),CompleteGraph(4),f,[CHQ_MONO,CHQ_MORPH$
[ 1, 2, 4, 3 ]
gap> f;
[ 1, 2, 4, 3 ]
```

95 ▶ `NumberOfCliques(` G `)` A
▶ `NumberOfCliques(` G, m `)` O

Returns the number of (maximal) cliques of G. In the second form, It stops computing cliques after m of them have been counted and returns m in case G has m or more cliques.

```
gap> NumberOfCliques(Icosahedron);
20
gap> NumberOfCliques(Icosahedron,15);
15
gap> NumberOfCliques(Icosahedron,50);
20
```

This implementation discards the cliques once counted hence, given enough time, it can compute the number of cliques of G even if the set of cliques does not fit in memory.

```
gap> NumberOfCliques(OctahedralGraph(30));
1073741824
```

–map

96 ► `OctahedralGraph(` n `)` F

Return the n-dimensional octahedron. This is the complement of n copies of K_2 (an edge). It is also the *(2n-2)*-regular graph on $2n$ vertices.

```
gap> OctahedralGraph(3);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 3, 4, 5, 6 ], [ 3, 4, 5, 6 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ],
[ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ] ] )
```

–map

97 ► `Octahedron` V

The 1-skeleton of Plato's octahedron.

```
gap> Octahedron;
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 3, 4, 5, 6 ], [ 3, 4, 5, 6 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ],
  [ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ] ] )
```

–map

98 ► `Order(` G `)` A

Returns the number of vertices, of graph G.

```
gap> Order(Icosahedron);
12
```

–map

99 ► `OrientedGraphs( )` C

Oriented Graphs are graphs which have arrows in only one direction between any two vertices.

100 ► `OutNeigh(` G, x `)` O

Returns the list of out-neighbors of x in G.

```
gap> tt:=CompleteGraph(5:GraphCategory:=OrientedGraphs);
Graph( Category := OrientedGraphs, Order := 5, Size := 10, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 3, 4, 5 ], [ 4, 5 ], [ 5 ], [  ] ] )
gap> InNeigh(tt,3);
[ 1, 2 ]
gap> OutNeigh(tt,3);
[ 4, 5 ]
```

–map

101 ► `ParachuteGraph` V

The complement of a `ParapluieGraph`; The suspension of a 4-path with a pendant vertex attached to the south pole.

```
gap> ParachuteGraph;
Graph( Category := SimpleGraphs, Order := 7, Size := 12, Adjacencies :=
[ [ 2 ], [ 1, 3, 4, 5, 6 ], [ 2, 4, 7 ], [ 2, 3, 5, 7 ], [ 2, 4, 6, 7 ],
  [ 2, 5, 7 ], [ 3, 4, 5, 6 ] ] )
```

–map

102 ► `ParapluieGraph` V

A 3-Fan graph with a 3-path attached to the universal vertex.

```
gap> ParapluieGraph;
Graph( Category := SimpleGraphs, Order := 7, Size := 9, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4, 5, 6, 7 ], [ 3, 5 ], [ 3, 4, 6 ], [ 3, 5, 7 ],
  [ 3, 6 ] ] )
```

–map

103 ► `PathGraph(` *n* `)` F

Returns the path graph on n vertices.

```
gap> PathGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3 ] ] )
```

–map

104 ► `PawGraph` V

The graph on 4 vertices, 4 edges and maximum degree 3: A triangle with a pendant vertex.

```
gap> PawGraph;
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2 ], [ 1, 3, 4 ], [ 2, 4 ], [ 2, 3 ] ] )
```

–map

105 ► `PowerGraph(` *G*`,` *e* `)` O

Returns the `DistanceGraph` of G using `[0, 1, ...,` *e*`]` as the list of distances. Note that the distance `0` in the list produces loops in the new graph only when the `TargetGraphCategory` admits loops.

```
gap> g:=PathGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 4, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4 ] ] )
gap> PowerGraph(g,1);
Graph( Category := SimpleGraphs, Order := 5, Size := 4, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4 ] ] )
gap> PowerGraph(g,1:GraphCategory:=Graphs);
Graph( Category := Graphs, Order := 5, Size := 13, Adjacencies :=
[ [ 1, 2 ], [ 1, 2, 3 ], [ 2, 3, 4 ], [ 3, 4, 5 ], [ 4, 5 ] ] )
```

–map

106 ► `PropertyMorphism(` *G1*`,` *G2*`,` *c* `)` O

Returns the first morphisms that is true for the list of checks c given graphs *G1* and *G2*.

```
gap> PropertyMorphism(CycleGraph(4),CompleteGraph(4),[CHQ_MONO,CHQ_MORPH]);
[ 1, 2, 3, 4 ]
```

107 ► `PropertyMorphisms(` *G1*`,` *G2*`,` *c* `)` O

Returns all morphisms that are true for the list of checks c given graphs *G1* and *G2*.

```
gap> PropertyMorphism(CycleGraph(4),CompleteGraph(4),[CHQ_MONO,CHQ_MORPH]);
[ [ 1, 2, 3, 4 ], [ 1, 2, 4, 3 ], [ 1, 3, 2, 4 ], [ 1, 3, 4, 2 ],
  [ 1, 4, 2, 3 ], [ 1, 4, 3, 2 ], [ 2, 1, 3, 4 ], [ 2, 1, 4, 3 ],
  [ 2, 3, 1, 4 ], [ 2, 3, 4, 1 ], [ 2, 4, 1, 3 ], [ 2, 4, 3, 1 ],
  [ 3, 1, 2, 4 ], [ 3, 1, 4, 2 ], [ 3, 2, 1, 4 ], [ 3, 2, 4, 1 ],
  [ 3, 4, 1, 2 ], [ 3, 4, 2, 1 ], [ 4, 1, 2, 3 ], [ 4, 1, 3, 2 ],
  [ 4, 2, 1, 3 ], [ 4, 2, 3, 1 ], [ 4, 3, 1, 2 ], [ 4, 3, 2, 1 ] ]
```

108 ▶ `QtfyIsSimple(` *G* `)` O

Returns how far is graph G from being simple.

109 ▶ `QuotientGraph(` *G*, *P* `)` O
▶ `QuotientGraph(` *G*, *L1*, *L2* `)` O

Returns the quotient graph of graph G given a vertex partition P, by identifying any two vertices in the same part. The vertices of the quotient graph are the parts in the partition P two of them being adjacent iff any vertex in one part is adjacent to any vertex in the other part. Singletons may be omited in P.

```
gap> g:=PathGraph(8);;
gap> QuotientGraph(g,[[1,5,8],[2],[3],[4],[6],[7]]);
Graph( Category := SimpleGraphs, Order := 6, Size := 7, Adjacencies :=
[ [ 2, 4, 5, 6 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ], [ 1, 6 ], [ 1, 5 ] ] )
gap> QuotientGraph(g,[[1,5,8]]);
Graph( Category := SimpleGraphs, Order := 6, Size := 7, Adjacencies :=
[ [ 2, 4, 5, 6 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ], [ 1, 6 ], [ 1, 5 ] ] )
```

In its second form, `QuotientGraph` identifies each vertex in list *L1*, with the corresponding vertex in list *L2*. *L1* and *L2* must have the same length, but any or both of them may have repetitions.

```
gap> g:=PathGraph(8);;
gap> QuotientGraph(g,[[1,7],[4,8]]);
Graph( Category := SimpleGraphs, Order := 6, Size := 7, Adjacencies :=
[ [ 2, 4, 6 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3, 5 ], [ 4, 6 ], [ 1, 5 ] ] )
gap> QuotientGraph(g,[1,4],[7,8]);
Graph( Category := SimpleGraphs, Order := 6, Size := 7, Adjacencies :=
[ [ 2, 4, 6 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3, 5 ], [ 4, 6 ], [ 1, 5 ] ] )
```

–map

110 ▶ `Radius(` *G* `)` A

Returns the minimal excentricity among the vertices of graph G.

```
gap> Radius(PathGraph(5));
2
```

–map

111 ▶ `RandomGraph(` *n*, *p* `)` F
▶ `RandomGraph(` *n* `)` F

Returns a random graph of order n taking the rational $p \in [0, 1]$ as the edge probability.

```
gap> RandomGraph(5,1/3);
Graph( Category := SimpleGraphs, Order := 5, Size := 2, Adjacencies :=
[ [ 5 ], [ 5 ], [  ], [  ], [ 1, 2 ] ] )
gap> RandomGraph(5,2/3);
Graph( Category := SimpleGraphs, Order := 5, Size := 6, Adjacencies :=
[ [ 4, 5 ], [ 3, 4, 5 ], [ 2, 4 ], [ 1, 2, 3 ], [ 1, 2 ] ] )
gap> RandomGraph(5,1/2);
Graph( Category := SimpleGraphs, Order := 5, Size := 4, Adjacencies :=
[ [ 2, 5 ], [ 1, 3, 5 ], [ 2 ], [  ], [ 1, 2 ] ] )
```

If p is ommited, the edge probability is taken to be 1/2.

```
gap> RandomGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 2, 3 ], [ 1 ], [ 1, 4, 5 ], [ 3, 5 ], [ 3, 4 ] ] )
gap> RandomGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 3, Adjacencies :=
[ [ 2, 5 ], [ 1, 4 ], [  ], [ 2 ], [ 1 ] ] )
```

–map

112 ► `RemoveEdges(` G, E `)` O

Returns a new graph created from graph G by removing the edges in list E.

```
gap> g:=CompleteGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
gap> RemoveEdges(g,[[1,2]]);
Graph( Category := SimpleGraphs, Order := 4, Size := 5, Adjacencies :=
[ [ 3, 4 ], [ 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
gap> RemoveEdges(g,[[1,2],[3,4]]);
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 3, 4 ], [ 3, 4 ], [ 1, 2 ], [ 1, 2 ] ] )
```

–map

113 ► `RemoveVertices(` G, V `)` O

Returns a new graph created from graph G by removing the vertices in list V.

```
gap> g:=PathGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 4, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4 ] ] )
gap> RemoveVertices(g,[3]);
Graph( Category := SimpleGraphs, Order := 4, Size := 2, Adjacencies :=
[ [ 2 ], [ 1 ], [ 4 ], [ 3 ] ] )
gap> RemoveVertices(g,[1,3]);
Graph( Category := SimpleGraphs, Order := 3, Size := 1, Adjacencies :=
[ [  ], [ 3 ], [ 2 ] ] )
```

–map

114 ► `RGraph` V

A square with two pendant vertices attached to the same vertex of the square.

```
gap> RGraph;
Graph( Category := SimpleGraphs, Order := 6, Size := 6, Adjacencies :=
[ [ 2 ], [ 1, 3, 5, 6 ], [ 2, 4 ], [ 3, 5 ], [ 2, 4 ], [ 2 ] ] )
```

–map

115 ▶ `RingGraph(` *Rng*, *elms* `)` O

Returns the graph G whose vertices are the elements of the ring *Rng* such that x is adjacent to y iff x+r=y for some r in *elms*.

```
gap> r:=FiniteField(8);Elements(r);
GF(2^3)
[ 0*Z(2), Z(2)^0, Z(2^3), Z(2^3)^2, Z(2^3)^3, Z(2^3)^4, Z(2^3)^5, Z(2^3)^6 ]
gap> RingGraph(r,[Z(2^3),Z(2^3)^4]);
Graph( Category := SimpleGraphs, Order := 8, Size := 8, Adjacencies :=
[ [ 3, 6 ], [ 5, 7 ], [ 1, 4 ], [ 3, 6 ], [ 2, 8 ], [ 1, 4 ], [ 2, 8 ],
  [ 5, 7 ] ] )
```

–map

116 ▶ `SetCoordinates(` *G*, *Coord* `)` O

Sets the coordinates of the vertices of *G*, which are used to draw *G* by `Draw(` *G* `)`.

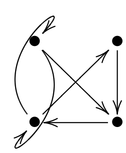
```
gap> g:=CycleGraph(4);;
gap> SetCoordinates(g,[[-10,-10 ],[-10,20],[20,-10 ], [20,20]]);
gap> Coordinates(g);
[ [ -10, -10 ], [ -10, 20 ], [ 20, -10 ], [ 20, 20 ] ]
```

–map

117 ▶ `SetDefaultGraphCategory(` *C* `)` F

Sets category C to be the default category for graphs. The default category is used, for instance, when constructing new graphs.

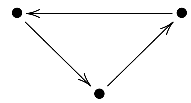
```
gap> SetDefaultGraphCategory(Graphs);
gap> g:=RandomGraph(4);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )
```



RandomGraph creates a random graphs belonging to the category graphs. The above graph has loops which are not permitted in simple graphs.

```
gap> SetDefaultGraphCategory(SimpleGraphs);
gap> g:=CopyGraph(g);
Graph( Category := SimpleGraphs, Order := 4, Size := 5, Adjacencies :=
[ [ 3, 4 ], [ 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
```

Now G is a simple graph.



118 ▶ `SimpleGraphs( )` C

Simple Graphs are graphs with no loops and undirected.

119 ▶ `Size(` G `)` A

Returns the number of edges of graph G.

```
gap> Size(Icosahedron);
30
```

–map

120 ▶ `SnubDisphenoid` V

The 1-skeleton of the 84th Johnson solid.

```
gap> SnubDisphenoid;
Graph( Category := SimpleGraphs, Order := 8, Size := 18, Adjacencies :=
[ [ 2, 3, 4, 5, 8 ], [ 1, 3, 6, 7, 8 ], [ 1, 2, 4, 6 ], [ 1, 3, 5, 6 ],
  [ 1, 4, 6, 7, 8 ], [ 2, 3, 4, 5, 7 ], [ 2, 5, 6, 8 ], [ 1, 2, 5, 7 ] ] )
```

–map

121 ▶ `SpikyGraph(` N `)` F

The spiky graph is constructed as follows: Take complete graph on N vertices, K_N, and then, for each the N subsets of *Vertices*(K_n) of order *N-1*, add an additional vertex which is adjacent precisely to this subset of *Vertices*(K_n).

```
gap> SpikyGraph(3);
Graph( Category := SimpleGraphs, Order := 6, Size := 9, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 4, 6 ], [ 1, 2, 5, 6 ], [ 1, 2 ], [ 1, 3 ],
  [ 2, 3 ] ] )
```

–map

122 ▶ `SunGraph(` N `)` F

Returns the N-Sun: A complete graph on N vertices, K_N, with a corona made with a zigzagging *2N*-cycle glued to a N-cyle of the K_N.

```
gap> SunGraph(3);
Graph( Category := SimpleGraphs, Order := 6, Size := 9, Adjacencies :=
[ [ 2, 6 ], [ 1, 3, 4, 6 ], [ 2, 4 ], [ 2, 3, 5, 6 ], [ 4, 6 ],
  [ 1, 2, 4, 5 ] ] )
gap> SunGraph(4);
Graph( Category := SimpleGraphs, Order := 8, Size := 14, Adjacencies :=
[ [ 2, 8 ], [ 1, 3, 4, 6, 8 ], [ 2, 4 ], [ 2, 3, 5, 6, 8 ], [ 4, 6 ],
  [ 2, 4, 5, 7, 8 ], [ 6, 8 ], [ 1, 2, 4, 6, 7 ] ] )
```

–map

123 ▶ `TargetGraphCategory( [`G`, ... ] )` F

Returns the category which will be used to process a list of graphs. If an option category has been given it will return that category. Otherwise it will behave as Function *GraphCategory* (6). See Section 2 for the relationship among categories.

124 ▶ `Tetrahedron` V

The 1-skeleton of Plato's tetrahedron.

```
gap> Tetrahedron;
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
```

–map

125 ▶ `TimesProduct(` G`,` H `)` O

Returns the times product of two graphs G and H, $G \times H$ (also known as the tensor product).

The times product is computed as follows:

For each pair of vertices $g \in G, h \in H$ we create a vertex (g, h). Given two such vertices (g, h) and (g', h') they are adjacent *iff* $g \sim g'$ and $h \sim h'$.

```
gap> g1:=PathGraph(3);g2:=CycleGraph(4);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
gap> g1g2:=TimesProduct(g1,g2);
Graph( Category := SimpleGraphs, Order := 12, Size := 16, Adjacencies :=
[ [ 6, 8 ], [ 5, 7 ], [ 6, 8 ], [ 5, 7 ], [ 2, 4, 10, 12 ], [ 1, 3, 9, 11 ],
  [ 2, 4, 10, 12 ], [ 1, 3, 9, 11 ], [ 6, 8 ], [ 5, 7 ], [ 6, 8 ], [ 5, 7 ] ] )
gap> VertexNames(g1g2);
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 1 ], [ 2, 2 ], [ 2, 3 ],
  [ 2, 4 ], [ 3, 1 ], [ 3, 2 ], [ 3, 3 ], [ 3, 4 ] ]
```

–map

126 ▶ `TrivialGraph` V

The one vertex graph.

```
gap> TrivialGraph;
Graph( Category := SimpleGraphs, Order := 1, Size := 0, Adjacencies :=
[ [  ] ] )
```

–map

127 ▶ `UndirectedGraphs( )` C

Undirected Graphs are graphs which have no directed arrows.

128 ▶ `UnitsRingGraph(` *Rng* `)` O

Returns the graph G whose vertices are the elements of *Rng* such that x is adjacent to y iff x+z=y for some unit z of *Rng*

```
gap> UnitsRingGraph(ZmodnZ(8));
Graph( Category := SimpleGraphs, Order := 8, Size := 16, Adjacencies :=
[ [ 2, 4, 6, 8 ], [ 1, 3, 5, 7 ], [ 2, 4, 6, 8 ], [ 1, 3, 5, 7 ],
  [ 2, 4, 6, 8 ], [ 1, 3, 5, 7 ], [ 2, 4, 6, 8 ], [ 1, 3, 5, 7 ] ] )
```

–map

129 ▶ `VertexDegree(` G, v `)` O

Returns the degree of vertex v in Graph G.

```
gap> g:=PathGraph(3);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
gap> VertexDegree(g,1);
1
gap> VertexDegree(g,2);
2
```

–map

130 ▶ `VertexDegrees(` G `)` O

Returns the list of degrees of the vertices in graph G.

```
gap> g:=GemGraph;
Graph( Category := SimpleGraphs, Order := 5, Size := 7, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 3, 5 ], [ 1, 4 ] ] )
gap> VertexDegrees(g);
[ 4, 2, 3, 3, 2 ]
```

–map

131 ▶ `VertexNames(` G `)` A

Return the list of names of the vertices of G. The vertices of a graph in YAGS are always $\{1, 2, \ldots, Order(G)\}$, but depending on how the graph was constructed, its vertices may have also some *names*, that help us identify the origin of the vertices. YAGS will always try to store meaninful names for the vertices. For example, in the case of the LineGraph, the vertex names of the new graph are the edges of the old graph.

```
gap> g:=LineGraph(DiamondGraph);
Graph( Category := SimpleGraphs, Order := 5, Size := 8, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4, 5 ], [ 1, 2, 5 ], [ 1, 2, 5 ], [ 2, 3, 4 ] ] )
gap> VertexNames(g);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 3, 4 ] ]
gap> Edges(DiamondGraph);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 3, 4 ] ]
```

–map

132 ▶ `Vertices(` G `)` O

Returns the list [1..Order(G)].

```
gap> Vertices(Icosahedron);
[ 1 .. 12 ]
```

–map

133 ▶ `WheelGraph(` N `)` O
▶ `WheelGraph(` N, *Radius* `)` O

In its first form `WheelGraph` returns the wheel graph on *N+1* vertices. This is the cone of a cycle: a central vertex adjacent to all the vertices of an N-cycle

```
WheelGraph(5);
gap> Graph( Category := SimpleGraphs, Order := 6, Size := 10, Adjacencies :=
[ [ 2, 3, 4, 5, 6 ], [ 1, 3, 6 ], [ 1, 2, 4 ], [ 1, 3, 5 ], [ 1, 4, 6 ],
 [ 1, 2, 5 ] ] )
```

In its second form, `WheelGraph` returns returns the wheel graph, but adding *Radius-1* layers, each layer is a new N-cycle joined to the previous layer by a zigzagging *2N*-cycle. This graph is a triangulation of the disk.

```
gap> WheelGraph(5,2);
Graph( Category := SimpleGraphs, Order := 11, Size := 25, Adjacencies :=
[ [ 2, 3, 4, 5, 6 ], [ 1, 3, 6, 7, 8 ], [ 1, 2, 4, 8, 9 ], [ 1, 3, 5, 9, 10 ],
  [ 1, 4, 6, 10, 11 ], [ 1, 2, 5, 7, 11 ], [ 2, 6, 8, 11 ], [ 2, 3, 7, 9 ],
  [ 3, 4, 8, 10 ], [ 4, 5, 9, 11 ], [ 5, 6, 7, 10 ] ] )
gap> WheelGraph(5,3);
Graph( Category := SimpleGraphs, Order := 16, Size := 40, Adjacencies :=
[ [ 2, 3, 4, 5, 6 ], [ 1, 3, 6, 7, 8 ], [ 1, 2, 4, 8, 9 ], [ 1, 3, 5, 9, 10 ],
  [ 1, 4, 6, 10, 11 ], [ 1, 2, 5, 7, 11 ], [ 2, 6, 8, 11, 12, 13 ],
  [ 2, 3, 7, 9, 13, 14 ], [ 3, 4, 8, 10, 14, 15 ], [ 4, 5, 9, 11, 15, 16 ],
  [ 5, 6, 7, 10, 12, 16 ], [ 7, 11, 13, 16 ], [ 7, 8, 12, 14 ],
  [ 8, 9, 13, 15 ], [ 9, 10, 14, 16 ], [ 10, 11, 12, 15 ] ] )
```

–map

Bibliography

[BK73] Coen Bron and Joep Kerbosch. Finding all cliques of an undirected graph–algorithm 457. *Communications of the ACM*, 16:575–577, 1973.

[Dra89] Feodor F. Dragan. *Centers of graphs and the Helly property (in Russian)*. PhD thesis, Moldava State University, Chisinău, Moldava, 1989.

[HK96] Johann Hagauer and Sandi Klavar. Clique-gated graphs. *Discrete Mathematics*, 161(13):143 – 149, 1996.

[Piz04] M. A. Pizaña. Distances and diameters on iterated clique graphs. *Discrete Appl. Math.*, 141(1-3):255–161, 2004.

[Szw97] Jayme L. Szwarcfiter. Recognizing clique-Helly graphs. *Ars Combin.*, 45:29–32, 1997.

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".

U

V

W