

YAGS

Yet Another Graph System

GAP4 Package

Version 0.8

by

R. Mac Kinney Romero

M. A. Pizaña López

Electrical Engineering Department

Universidad Autonoma Metropolitana

email: rene,map@xanum.uam.mx

July 2014

Contents

1	Basics	3
1.1	Using YAGS	3
1.2	Definition of graphs	4
1.3	A taxonomy of graphs	4
1.4	Creating Graphs	6
1.5	Transforming graphs	8
1.6	Experimenting on graphs	8
2	Categories	9
2.1	Graph Categories	9
2.2	Default Category	11
3	Constructing graphs	12
3.1	Primitives	12
3.2	Families	16
3.3	Unary operations	22
3.4	Binary operations	23
4	Inspecting Graphs	27
4.1	Attributes and properties of graphs	27
4.2	Information about graphs	29
4.3	Distances	30
5	Morphisms of Graphs	33
5.1	Core Operations	33
5.2	Morphisms	34
6	Other Functions	35
	Bibliography	57
	Index	58

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 and 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
```

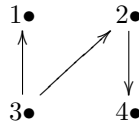
Returns a Random Graph of order <n>. The first form additionally takes a parameter <p>, the probability of an edge to exist. A probability 1 will return a Complete Graph and a probability 0 a Discrete Graph.

```
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 $M[i, j] = \text{true}$ 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 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.



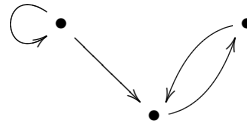
YAGS handles graphs that have arrows, edges and loops. Graphs that, for instance, have multiple arrows between vertices are not handled by YAGS.



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

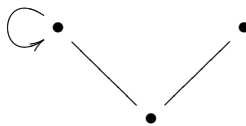
- *Graphs*: graphs with no particular property.



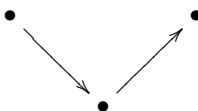
- *Loopless*: graphs with no loops.



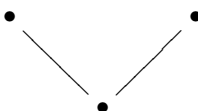
- *Undirected*: graphs with no arrows but only edges.



- *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

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

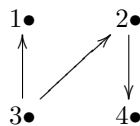
and the adjacency matrix

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

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 ] ] )
```

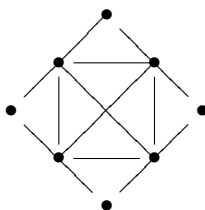
that yields the graph *h*



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

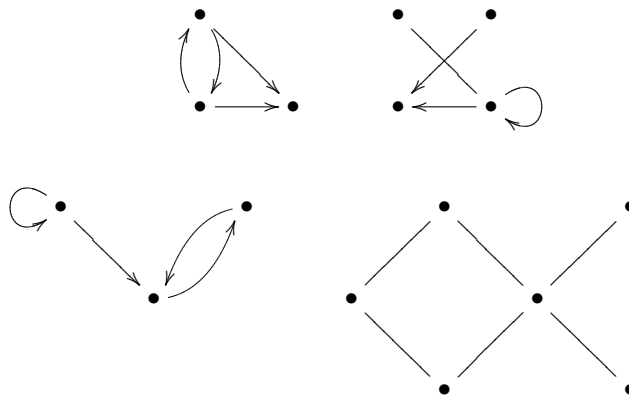
2.1 Graph Categories

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()`

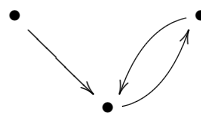
C

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.

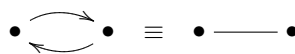


3 ► `UndirectedGraphs()`

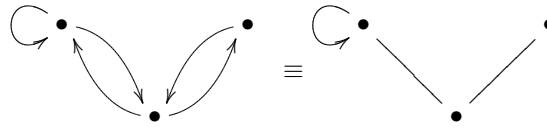
C

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 arrow (j, i) .



Undirected graphs have no arrows but only edges.

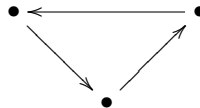


4 ► `OrientedGraphs()`

C

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

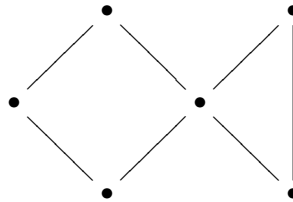
Oriented graphs have no edges but only arrows.



5 ► `SimpleGraphs()`

C

Simple Graphs are graphs with no loops and undirected.



The following figure shows the relationships among categories.

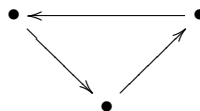
Graphs

Loopless Undirected

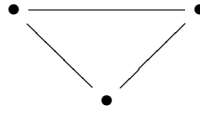
Oriented SimpleGraphs

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.

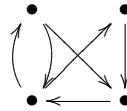
2.2 Default Category

There are several ways to specify the category in which a new graph will be created. There exists a *Default-Category* 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.

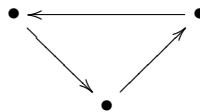
```
SetDefaultGraphCategory(Graphs);
G:=RandomGraph(4);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )
```



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

```
SetDefaultGraphCategory(SimpleGraphs);
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

Creates a graph 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

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

```
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

Creates 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

Creates a new graph 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

Creates a new graph 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

Creates a new graph 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

The clique graph of graph G , $K(G)$. The additional parameter m stops the operation when a maximum of m cliques have been found.

```

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> CliqueGraph(G);
Graph( Category := SimpleGraphs, Order := 5, Size := 8, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 4 ], [ 1, 2, 5 ], [ 1, 2, 5 ], [ 1, 3, 4 ] ] )

```

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 a Discrete Graph of order n . A discrete graph is a graph where vertices are unconnected.

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

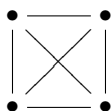


4-Discrete Graph

2 ► `CompleteGraph(n)` F

Returns a 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 ] ] )
```



4-Complete Graph

3 ► `PathGraph(n)` F

Returns a Path Graph of order n . A path graph is a graph connected forming a path.

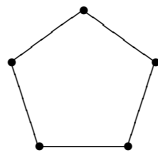
```
gap> PathGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3 ] ] )
```

4-Path Graph 

4 ► `CycleGraph(n)` F

Returns a Cycle Graph of order n . A cycle graph is a path graph where the vertices at the ends are connected.

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



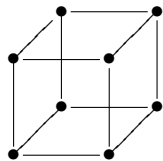
5-Cycle Graph

5 ► `CubeGraph(n)` F

Returns a Cube Graph of order n . A cube graph is a graph where each vertex has degree n .


```
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 ] ] )
```

3-Cube Graph

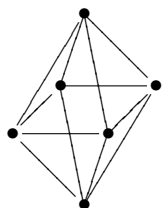


6 ► OctahedralGraph(n)

F

```
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 ] ] )
```

3-Octahedral Graph



7 ► JohnsonGraph(n, r)

F

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

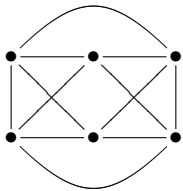
$$V(J(n, r)) = \{X \subset \{1, \dots, n\} \mid |X| = r\}$$

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

$$X \sim X' \text{ iff } |X \cap X'| = r - 1.$$

```
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 ] ] )
```

4,2-Johnson Graph



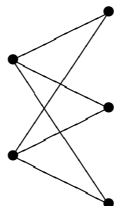
8 ► CompleteBipartiteGraph(n , m)

F

Returns a Complete Bipartite Graph of order $n + m$. A complete bipartite graph is the result of joining two Discrete graphs and adding edges to connect all vertices of each graph.

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

2,3-Complete Bipartite Graph

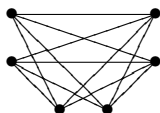
9 ► CompleteMultipartiteGraph(n_1 , n_2 [, n_3 ...])

F

Returns a Complete Multipartite Graph of order $n_1 + n_2 + \dots$. A complete multipartite graph is the result of joining Discrete graphs and adding edges to connect all vertices of each graph.

```
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 ] ] )
```

2,2,2-Complete Multipartite Graph

10 ► 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 ] ] )
```

5-Random Graph

11 ► WheelGraph(N)

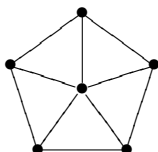
O

► WheelGraph(N , $Radius$)

O

```
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 ] ] )
```

Wheel Graph of Order 5

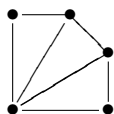


12 ► FanGraph(*N*)

F

```
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 ] ] )
```

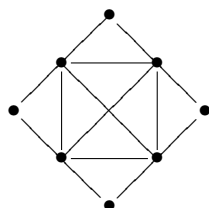
4-Fan Graph

13 ► SunGraph(*N*)

F

```
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 ] ] )
```

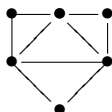
4-Sun Graph

14 ► SpikyGraph(*N*)

F

```
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 ] ] )
```

3-Spiky Graph



15 ► TrivialGraph

V

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

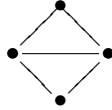
Trivial Graph •

16 ► DiamondGraph

V

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

Diamond Graph

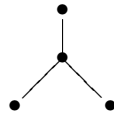


17 ► ClawGraph

V

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

Claw Graph

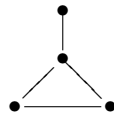


18 ► PawGraph

V

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

Paw Graph

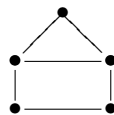


19 ► HouseGraph

V

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

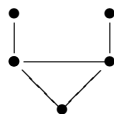
House Graph



20 ► BullGraph

V

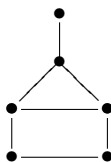
Bull Graph



21 ► AntennaGraph

V

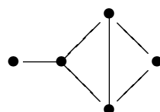
Antenna Graph



22 ► KiteGraph

V

Kite Graph

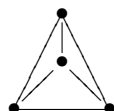


23 ► Tetrahedron

V

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

Tetrahedron

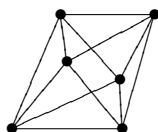


24 ► Octahedron

V

```
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 ] ] )
```

Octahedron

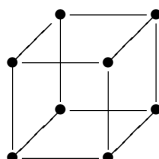


25 ► Cube

V

```
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 ] ] )
```

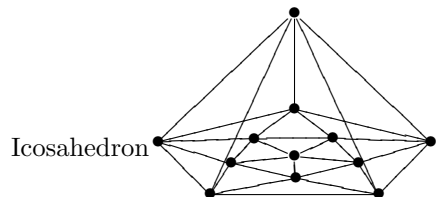
Cube Graph



26 ► Icosahedron

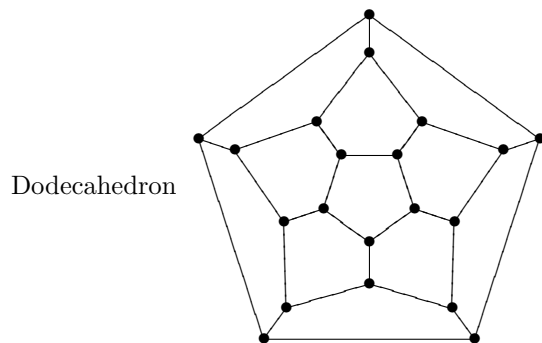
V

```
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 ] ] )
```



27 ► Dodecahedron

V



3.3 Unary operations

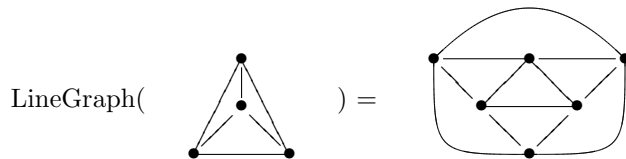
These are operations that can be performed over graphs.

1 ► LineGraph(<G>)

O

Returns the line graph 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 ] ] )
```

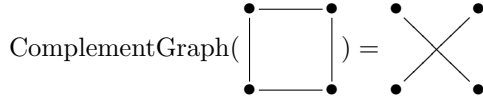


2 ► ComplementGraph(<G>)

A

Computes the complement of graph iG_i . The complement of a graph is created as follows: Create a graph iG'_i with same vertices of iG_i . For each $ix_i, iy_i \in iG_i$ if $ix_i \approx iy_i$ in iG_i then $ix_i \sim iy_i$ in iG'_i .

```
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 ] ] )
```



3 ► QuotientGraph(<G>, <P>)

O

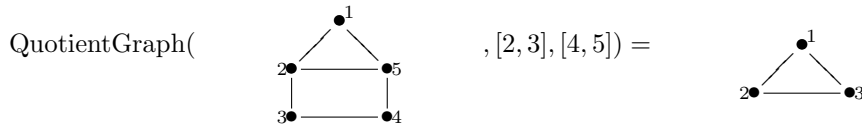
► QuotientGraph(<G>, <L1>, <L2>)

O

Returns the quotient graph of graph iG_i given a vertex partition iP_i , by identifying any two vertices in the same part. The vertices of the quotient graph are the parts in the partition iP_i two of them being adjacent iff any two of the vertices in the parts are adjacent in iG_i . Singletons may be omitted in P.

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



3.4 Binary operations

These are binary operations that can be performed over graphs.

1 ► BoxProduct(G, H)

O

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

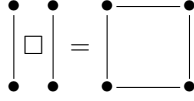
The box product is calculated as follows:

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> G:=GraphByAdjMatrix([[false,true],[true,false]]);
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies :=
[ [ 2 ], [ 1 ] ] )
gap> BoxProduct(G,G);
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 3 ], [ 1, 4 ], [ 1, 4 ], [ 2, 3 ] ] )

```



2► TimesProduct(G, H)

O

Returns the times product of two graphs G and H , $G \times H$.

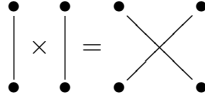
The times product is computed as follows:

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> G:=GraphByAdjMatrix([[false,true],[true,false]]);
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies :=
[ [ 2 ], [ 1 ] ] )
gap> TimesProduct(G,G);
Graph( Category := SimpleGraphs, Order := 4, Size := 2, Adjacencies :=
[ [ 4 ], [ 3 ], [ 2 ], [ 1 ] ] )

```



3► BoxTimesProduct(G, H)

O

Returns the box times product of two graphs G and H , $G \boxtimes H$.

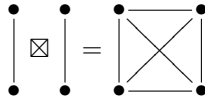
The box times product is calculated as follows:

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> G:=GraphByAdjMatrix([[false,true],[true,false]]);
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies :=
[ [ 2 ], [ 1 ] ] )
gap> G:=BoxTimesProduct(G,G);
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )

```



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

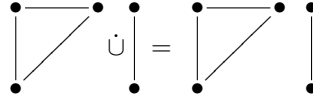
4► DisjointUnion(G, H)

O

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

A disjoint union of graphs is obtained by combining both graphs in a disjoint graph.

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



5 ► Join(G , H)

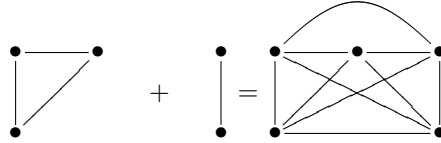
O

Returns the result of joining graph G and H , $G + H$.

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 $h \in H$. Finally, for each vertex $g \in G$ we add an edge to each vertex $h \in H$.

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



6 ► GraphSum(G , L)

O

Returns the GraphSum of G and a list of graphs L .

The GraphSum is computed as follows:

Given G and a list of graphs $L = L_1, \dots, L_n$, if G is the trivial graph the result is L_1 . Otherwise we take $g_1, g_2 \in G$. If $g_1 \sim g_2$ we compute $L_1 + L_2$ and the DisjointUnion in other case. We repeat this process for every g_i, g_{i+1} until $i + 1 = n$.

If a graph is not given in a particular element of the list the trivial graph will be used, *e.g.* $[H, , J,] \equiv [H, T, J, T]$ where T is the trivial graph.

```

gap> G:=RandomGraph(4);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )
gap> H:=RandomGraph(4);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )
gap> GraphSum(G,H);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )

```

7 ► `Composition(G, H)`

O

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

A composition of graphs is obtained by calculating the `GraphSum` of G with H ,

$$G[H] = \text{GraphSum}(G, [H, \dots, H]).$$

```

gap> G:=RandomGraph(4);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )
gap> H:=RandomGraph(4);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )
gap> Composition(G,H);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )

```

4

Inspecting Graphs

4.1 Attributes and properties of graphs

The following are functions to obtain attributes 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 a graph G . The vertices of a graph in YAGS are always $\{1, 2, \dots, \text{Order}(G)\}$, 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 meaningful 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)` A
 ▶ `QtifyIsCompleteGraph(G)` P

The attribute form is **true** if graph G is complete. The property form measures how far graph G is from being complete.

- 6 ▶ `IsLoopless(G)` A
 ▶ `QtifyIsLoopless(G)` P

The attribute form is **true** if graph G has no loops. The property form measures how far graph G is from being loopless, *i.e.* the number of loops in G .

- 7 ▶ `IsUndirected(G)` A
 ▶ `QtifyIsUndirected(G)` P

The attribute form is **true** if graph G has only edges and no arrows. The property form measures how far graph G is from being undirected, *i.e.* the number of arrows in G .

- 8 ▶ `IsOriented(G)` A
 ▶ `QtifyIsOriented(G)` P

The attribute form is **true** if graph G has only arrows. The property form measures how far graph G is from being oriented, *i.e.* the number of edges in G .

- 9 ▶ `CliqueNumber(G)` A

The order of the largest clique in G , $\omega(G)$.

```
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
```

- 10 ▶ `Cliques(G)` A
 ▶ `Cliques(G, m)` O

The set of all cliques in graph G using the Bron-Kerbosch algorithm. The additional parameter m stops the operation when a maximum of m cliques have been found.

Each clique is represented by the set of vertices in G that belong to the clique. A set is represented as a list.

```
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> Cliques(G);
[ [ 2, 4, 6, 8 ], [ 2, 4, 3 ], [ 2, 1, 8 ], [ 5, 4, 6 ], [ 7, 6, 8 ] ]
```

- 11 ▶ `IsCliqueHelly(G)` A
 ▶ `QtifyIsCliqueHelly(G)` P

The attribute form is **true** if all cliques of graph G satisfy the *Helly* property. The property form measures how far graph G is from being cliquehelly, *i.e.* the number of non-Helly cliques in G .

The Helly property is defined as follows:

A family \mathcal{F} of sets satisfies the Helly property if

$$\begin{aligned} & \forall \mathcal{K} \subseteq \mathcal{F} \\ & \forall x_1, x_2 \in \mathcal{K} \\ & x_1 \cap x_2 \neq \emptyset \Rightarrow \bigcap \mathcal{K} \neq \emptyset \end{aligned}$$

The algorithm used to compute the Helly property is by Jayme

```
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
gap> QtfyIsCliqueHelly(G);
1
```

4.2 Information about graphs

The following functions give information regarding graphs.

1 ► `IsSimple(G)` O

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

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 graphs 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 ] ]
```

4.3 Distances

These are functions that measure distances between graphs.

1 ► `Distance(G, x, y)` O

Returns the minimal number of edges that connect vertices x and y .

$$d_G(x, y)$$

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

2 ► DistanceMatrix(*G*)

A

Returns the matrix of distances for all vertices in *G*. The matrix is asymmetric if the graph is directed. An entry in the matrix of ∞ means there is no path between the vertices. Floyd's algorithm is used to compute the matrix.

```
gap> G:=CycleGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 2, 5 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 1, 4 ] ] )
gap> DistanceMatrix(G);
[ [ 0, 1, 2, 2, 1 ], [ 1, 0, 1, 2, 2 ], [ 2, 1, 0, 1, 2 ], [ 2, 2, 1, 0, 1 ],
[ 1, 2, 2, 1, 0 ] ]
```

3 ► Diameter(*G*)

A

The diameter of a graph *G* is the maximum distance for any two vertices in *G*.

$$\max\{d_G(x,y) | x,y \in V(G)\}$$

```
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
```

4 ► Excentricity(*G*, *x*)

F

Returns the distance from a vertex *x* in graph *G* to the furthest away vertex in *G*.

$$\max\{d_G(x,y) | y \in V(G)\}$$

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

5 ► Radius(*G*)

A

Returns the minimal excentricity among the vertices of graph *G*.

$$\min\{Excentricity(G,x) | x \in V(G)\}$$

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

6 ► Distances(*G*, *A*, *B*)

O

Given two subsets of vertices *A*, *B* of graph *G* returns the list of distances for every pair in the cartesian product of *A* and *B*.

$$[d_G(x,y) | (x,y) \in A \times B]$$

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

7 ► DistanceSet(*G*, *A*, *B*)

O

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

$$\{d_G(x, y) | (x, y) \in A \times B\}$$

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

8 ► DistanceGraph(*G*, *D*)

O

Given a graph *G* and list of Distances *D* returns the graph constructed using the vertices of *G* where two vertices are adjacent iff the distance between them is in 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 ] ] )
```

9 ► PowerGraph(*G*, *e*)

O

Returns the Distance graph of *G* using as a list of distances $[0, 1, \dots, e]$. Note that the distance 0 is used only if *G* has loops.

$$G^n = \text{DistanceGraph}(G, [0, 1, \dots, e])$$

```
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> PowerGraph(G, 3);
Graph( Category := SimpleGraphs, Order := 8, Size := 28, Adjacencies :=
[ [ 2, 3, 4, 5, 6, 7, 8 ], [ 1, 3, 4, 5, 6, 7, 8 ], [ 1, 2, 4, 5, 6, 7, 8 ],
[ 1, 2, 3, 5, 6, 7, 8 ], [ 1, 2, 3, 4, 6, 7, 8 ], [ 1, 2, 3, 4, 5, 7, 8 ],
[ 1, 2, 3, 4, 5, 6, 8 ], [ 1, 2, 3, 4, 5, 6, 7 ] ] )
```


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 $\varphi : G \rightarrow H$. 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

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

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

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

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

$$|H| = |G|.$$

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 .

Iso An isomorphism is a bimorphism which is also complete.

Additionally 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

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

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

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

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

6

Other Functions

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

1 ► AddEdges(G , E)

O

Creates a new graph 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

6 ► AntennaGraph V

FIXME: Declaration: AutomorphismGroup

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

8 ► BackTrackBag(*opts*, *chk*, *done*, *extra*) O

9 ► Basement(*G*, *KnG*, *x*) O

► Basement(*G*, *KnG*, *V*) O

The operation calculates the basement

Given a graph G and Q a subset of the vertices in G the basement of G

```
G:=RandomGraph(4);
Order(G);
4
```

10 ► BoxProduct(*G*, *H*) O

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

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> G:=GraphByAdjMatrix([[false,true],[true,false]]);
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies :=
[ [ 2 ], [ 1 ] ] )
gap> BoxProduct(G,G);
Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
[ [ 2, 3 ], [ 1, 4 ], [ 1, 4 ], [ 2, 3 ] ] )
```

11 ► BoxTimesProduct(*G*, *H*) O

Returns the box times product of two graphs G and H , $G \boxtimes H$.

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> G:=GraphByAdjMatrix([[false,true],[true,false]]);
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies :=
[ [ 2 ], [ 1 ] ] )
gap> G:=BoxTimesProduct(G,G);
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
```

12 ► BullGraph V

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.

14 ► ChairGraph V

15 ► Circulant(n , $jumps$) O

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

16 ► ClawGraph V

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

17 ► CliqueGraph(G) A

► CliqueGraph(G , m) O

The clique graph of graph G , $K(G)$. The additional parameter m stops the operation when a maximum of m cliques have been found.

```
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> CliqueGraph(G);
Graph( Category := SimpleGraphs, Order := 5, Size := 8, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 4 ], [ 1, 2, 5 ], [ 1, 2, 5 ], [ 1, 3, 4 ] ] )
```

18 ► CliqueNumber(G) A

The order of the largest clique in G , $\omega(G)$.

```
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
```

19 ► Cliques(G) A

► Cliques(G , m) O

The set of all cliques in graph G using the Bron-Kerbosch algorithm. The additional parameter m stops the operation when a maximum of m cliques have been found.

Each clique is represented by the set of vertices in G that belong to the clique. A set is represented as a list.

```
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> Cliques(G);
[ [ 2, 4, 6, 8 ], [ 2, 4, 3 ], [ 2, 1, 8 ], [ 5, 4, 6 ], [ 7, 6, 8 ] ]
```

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 $x \not\sim y$ 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 ] ] )

```

21 ► CompleteBipartiteGraph(n , m)

F

Returns a Complete Bipartite Graph of order $n + m$. A complete bipartite graph is the result of joining two Discrete graphs and adding edges to connect all vertices of each graph.

```

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

```

22 ► CompleteGraph(n)

F

Returns a 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 ] ] )

```

23 ► CompleteMultipartiteGraph($n1$, $n2$ [, $n3$...])

F

Returns a Complete Multipartite Graph of order $n1 + n2 + \dots$. A complete multipartite graph is the result of joining Discrete graphs and adding edges to connect all vertices of each graph.

```

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 ] ] )

```

24 ► CompletesOfGivenOrder(G , o)

O

This operation finds all complete graphs 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 ] ]

```

25 ► Composition(G , H)

O

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

A composition of graphs is obtained by calculating the GraphSum of G with H ,

$$G[H] = \text{GraphSum}(G, [H, \dots, H]).$$

```

gap> G:=RandomGraph(4);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )
gap> H:=RandomGraph(4);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )
gap> Composition(G,H);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )

```

26 ► Coordinates(*G*)

O

Gets the coordinates of the vertices of *G*, which are used to 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 ] ]

```

27 ► CopyGraph(*G*)

O

Creates 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 ► QuadraticRingGraph(*Rng*)

O

Returns the graph *G* whose vertices are the elements of *Rng* such that *x* is adjacent to *y* iff $x+z^2=y$ for some *z* in *Rng*.

29 ► Cube

V

```

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 ] ] )

```

30 ► CubeGraph(*n*)

F

Returns a Cube Graph of order *n*. A cube graph is a graph where each vertex has degree *n*.

```
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 ] ] )
```

31 ► CycleGraph(*n*) F

Returns a Cycle Graph of order *n*. A cycle graph is a path graph where the vertices at the ends are connected.

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

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

33 ► DartGraph V

34 ► DeclareQtifyProperty(*N*, *F*) F

Declares a quantifiable property named *N* for filter *F*. A quantifiable property is a property that can be measured according to some metric. This Declaration actually declares two functions: a boolean property *N* and an integer property Qtfy*N*. The user must provide the method *N*(*O*, *qtfy*) where *qtfy* is a boolean that tells the method whether to quantify the property or simply return a boolean stating if the property is 'true' or 'false'.

```
gap> DeclareQtifyProperty("Is2Regular",Graphs);
gap> InstallMethod(Is2Regular,"for graphs",true,[Graphs,IsBool],0,
> function(G,qtfy)
>   local m;
>   m:=Length(Filtered(VertexDegrees(G),x->x<>2));
>   if qtfy then
>     return m;
>   else
>     return (m=0);
>   fi;
> end);
```

35 ► Diameter(*G*) A

The diameter of a graph *G* is the maximum distance for any two vertices in *G*.

$$\max\{d_G(x,y)|x,y \in V(G)\}$$

```
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
```

36 ► DiamondGraph V

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

37 ► DiscreteGraph(*n*) F

Returns a Discrete Graph of order *n*. A discrete graph is a graph where vertices are unconnected.


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

38 ► `DisjointUnion(G, H)`

O

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

A disjoint union of graphs is obtained by combining both graphs in a disjoint graph.

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

39 ► `Distance(G, x, y)`

O

Returns the minimal number of edges that connect vertices x and y .

$$d_G(x, y)$$

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

40 ► `DistanceGraph(G, D)`

O

Given a graph G and list of Distances D returns the graph constructed using the vertices of G where two vertices are adjacent iff the distance between them is in 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 ] ] )
```

41 ► `DistanceMatrix(G)`

A

Returns the matrix of distances for all vertices in G . The matrix is asymmetric if the graph is directed. An entry in the matrix of ∞ means there is no path between the vertices. Floyd's algorithm is used to compute the matrix.

```

gap> G:=CycleGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 2, 5 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 1, 4 ] ] )
gap> DistanceMatrix(G);
[ [ 0, 1, 2, 2, 1 ], [ 1, 0, 1, 2, 2 ], [ 2, 1, 0, 1, 2 ], [ 2, 2, 1, 0, 1 ],
  [ 1, 2, 2, 1, 0 ] ]

```

42 ► Distances(*G*, *A*, *B*) O

Given two subsets of vertices *A*, *B* of graph *G* returns the list of distances for every pair in the cartesian product of *A* and *B*.

$$[d_G(x, y) | (x, y) \in A \times B]$$

```

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

```

43 ► DistanceSet(*G*, *A*, *B*) O

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

$$\{d_G(x, y) | (x, y) \in A \times B\}$$

```

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

```

44 ► Dodecahedron V

45 ► DominoGraph V

46 ► Draw(*G*) O

Takes a graph *G* and makes a drawing of it in a separate window. The user can then modify the drawing and finally the coordinates of the vertices of *G* used for the drawing, are updated into the graph *G*.

```

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 ] ]

```

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" ] )
```

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 the furthest away vertex in G .

$$\max\{d_G(x, y) | y \in V(G)\}$$

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

50 ► `FanGraph(N)` F

```
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 ] ] )
```

51 ► `FishGraph` V

52 ► `GemGraph` V

53 ► `Graph(R)` O

Creates a graph 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 the 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

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

```

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 GraphSum of *G* and a list of graphs *L*.

The GraphSum is computed as follows:

Given *G* and a list of graphs $L = L_1, \dots, L_n$, if *G* is the trivial graph the result is L_1 . Otherwise we take $g_1, g_2 \in G$. If $g_1 \sim g_2$ we compute $L_1 + L_2$ and the DisjointUnion in other case. We repeat this process for every g_i, g_{i+1} until $i + 1 = n$.

If a graph is not given in a particular element of the list the trivial graph will be used, *e.g.* $[H, , J,] \equiv [H, T, J, T]$ where *T* is the trivial graph.

```

gap> G:=RandomGraph(4);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )
gap> H:=RandomGraph(4);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )
gap> GraphSum(G,H);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )

```

62 ► **GraphToRaw**(*filename*, *G*)

O

Converts a Yags graph *G* into a raw format (vertices, coordinates and adjacency matrix) and writes the converted data to the file *filename*. For use of the external program 'draw'.

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

63 ► `GraphUpdateFromRaw(filename, G)` O

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

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 $\{\{act(v, g), act(u, g)\} : g \in Grp, \{u, v\} \in Edges(G)\}$.

If act is omitted, the standard GAP action 'OnPoints' is used.

65 ► `HouseGraph` V

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

66 ► `Icosahedron` V

```
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 ] ] )
```

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, v)` O

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
```

72 ► IsCliqueGated(*G*, *qtfy*)

O

```
G:=RandomGraph(4);
Order(G);
4
```

73 ► IsCliqueHelly(*G*)

A

► QtfyIsCliqueHelly(*G*)

P

The attribute form is **true** if all cliques of graph *G* satisfy the *Helly* property. The property form measures how far graph *G* is from being cliquehelly, *i.e.* the number of non-Helly cliques in *G*.

The Helly property is defined as follows:

A family \mathcal{F} of sets satisfies the Helly property if

$$\begin{aligned} & \forall \mathcal{K} \subseteq \mathcal{F} \\ & \forall x_1, x_2 \in \mathcal{K} \\ & x_1 \cap x_2 \neq \emptyset \Rightarrow \bigcap \mathcal{K} \neq \emptyset \end{aligned}$$

The algorithm used to compute the Helly property is by Jayme

```
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
gap> QtfyIsCliqueHelly(G);
1
```

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)` A
 ▶ `QtifyIsCompleteGraph(G)` P
- The attribute form is **true** if graph G is complete. The property form measures how far graph G is from being complete.
- 76 ▶ `IsDiamondFree(G, qtfy)` O
- 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
- 79 ▶ `IsLoopless( G )` A  
 ▶ `QtifyIsLoopless( G )` P
- The attribute form is **true** if graph  $G$  has no loops. The property form measures how far graph  $G$  is from being loopless, *i.e.* the number of loops in  $G$ .
- 80 ▶ `IsoMorphism( G, H )` O
- 81 ▶ `IsoMorphisms( G, H )` O
- 82 ▶ `IsOriented( G )` A  
 ▶ `QtifyIsOriented( G )` P
- The attribute form is **true** if graph  $G$  has only arrows. The property form measures how far graph  $G$  is from being oriented, *i.e.* the number of edges in  $G$ .
- 83 ▶ `IsSimple( G )` O
- Returns **true** if the graph  $G$  is simple regardless of its category.
- 84 ▶ `IsTournament( G )` O
- 85 ▶ `IsTransitiveTournament( G )` O
- 86 ▶ `IsUndirected( G )` A  
 ▶ `QtifyIsUndirected( G )` P
- The attribute form is **true** if graph  $G$  has only edges and no arrows. The property form measures how far graph  $G$  is from being undirected, *i.e.* the number of arrows in  $G$ .
- 87 ▶ `JohnsonGraph( n, r )` F
- Returns a Johnson Graph  $J(n, r)$ . A Johnson Graph is a graph constructed as follows. Each vertex represents a subset of the set  $\{1, \dots, n\}$  with cardinality  $r$ .

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

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

$$X \sim X' \text{ iff } |X \cap X'| = r - 1.$$



```
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]])
```

88 ► Join(  $G$ ,  $H$  )

O

Returns the result of joining graph  $G$  and  $H$ ,  $G + H$ .

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  $h \in H$ . Finally, for each vertex  $g \in G$  we add an edge to each vertex  $h \in H$ .

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

89 ► KiteGraph

V

90 ► LineGraph(  $G$  )

O

Returns the line graph 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]])
```

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

94 ► `NextIsoMorphism( G, H, morph )` O

95 ► `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]
```

96 ► `NumberOfCliques( G )` A

► `NumberOfCliques( G, m )` O

FIXME: Document!

97 ► `OctahedralGraph( n )` F

```
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]])
```

98 ► `Octahedron` V

```
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]])
```

99 ► `Order( G )` A

Returns the number of vertices, of graph *G*.

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

—map

100 ► `OrientedGraphs( )` C

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

|       |                               |   |
|-------|-------------------------------|---|
| 101 ▶ | <code>OutNeigh( G, v )</code> | O |
| 102 ▶ | <code>ParachuteGraph</code>   | V |
| 103 ▶ | <code>ParapluieGraph</code>   | V |
| 104 ▶ | <code>PathGraph( n )</code>   | F |

Returns a Path Graph of order  $n$ . A path graph is a graph connected forming a path.

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

|       |                       |   |
|-------|-----------------------|---|
| 105 ▶ | <code>PawGraph</code> | V |
|-------|-----------------------|---|

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

|       |                                 |   |
|-------|---------------------------------|---|
| 106 ▶ | <code>PowerGraph( G, e )</code> | O |
|-------|---------------------------------|---|

Returns the Distance graph of  $G$  using as a list of distances  $[0,1,\dots,e]$ . Note that the distance 0 is used only if  $G$  has loops.

$$G^n = \text{DistanceGraph}(G, [0, 1, \dots, e])$$

```
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> PowerGraph(G,3);
Graph(Category := SimpleGraphs, Order := 8, Size := 28, Adjacencies :=
[[2, 3, 4, 5, 6, 7, 8], [1, 3, 4, 5, 6, 7, 8], [1, 2, 4, 5, 6, 7, 8],
[1, 2, 3, 5, 6, 7, 8], [1, 2, 3, 4, 6, 7, 8], [1, 2, 3, 4, 5, 7, 8],
[1, 2, 3, 4, 5, 6, 8], [1, 2, 3, 4, 5, 6, 7]])
```

|       |                                            |   |
|-------|--------------------------------------------|---|
| 107 ▶ | <code>PropertyMorphism( G1, G2, c )</code> | 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]
```

|       |                                             |   |
|-------|---------------------------------------------|---|
| 108 ▶ | <code>PropertyMorphisms( G1, G2, c )</code> | 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]]
```

|       |                                |   |
|-------|--------------------------------|---|
| 109 ▶ | <code>QttyIsSimple( G )</code> | O |
|-------|--------------------------------|---|

Returns how far is graph  $G$  from being simple.

110 ► `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 two of the vertices in the parts are adjacent in  $G$ . Singletons may be omitted in  $P$ .

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

111 ► `Radius( G )` A

Returns the minimal excentricity among the vertices of graph  $G$ .

$$\min\{Excentricity(G,x)|x \in V(G)\}$$

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

112 ► `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]])
```

113 ► `RemoveEdges( G, E )` O

Creates a new graph 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

114 ► `RemoveVertices( G, V )` O

Creates a new graph 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

115 ► RGraph V

116 ► 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  $elms$ .

117 ► SetCoordinates(  $G$ , *Coord* ) O

Sets the coordinates of the vertices of  $G$ , which are used to 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]]

```

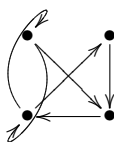
118 ► SetDefaultGraphCategory(  $C$  ) F

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

```

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

```



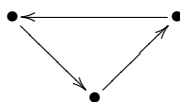
RandomGraph creates a random graphs belonging to the category graphs. The above graph has loops which are not permitted in simple graphs.

```

SetDefaultGraphCategory(SimpleGraphs);
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.



119 ► `SimpleGraphs( )` C

Simple Graphs are graphs with no loops and undirected.

120 ► `Size( G )` A

Returns the number of edges of graph  $G$ .

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

–map

121 ► `SnubDisphenoid` V

```
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]])
```

122 ► `SpikyGraph( N )` F

```
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]])
```

123 ► `SunGraph( N )` F

```
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]])
```

124 ► `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.

125 ► `Tetrahedron` V

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

126 ► `TimesProduct( G, H )` O

Returns the times product of two graphs  $G$  and  $H$ ,  $G \times H$ .

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> G:=GraphByAdjMatrix([[false,true],[true,false]]);
Graph(Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies :=
[[2], [1]])
gap> TimesProduct(G,G);
Graph(Category := SimpleGraphs, Order := 4, Size := 2, Adjacencies :=
[[4], [3], [2], [1]])

```

## 127 ► TrivialGraph

V

```

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

```

## 128 ► UndirectedGraphs( )

C

Undirected Graphs are graphs which have no directed arrows.

129 ► 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  $Rng$

130 ► 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

131 ► 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

132 ► VertexNames(  $G$  )

A

Return the list of names of the vertices of a graph  $G$ . The vertices of a graph in YAGS are always  $\{1, 2, \dots, \text{Order}(G)\}$ , 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 meaningful 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

133 ► **Vertices( *G* )** O

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

```

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

```

—map

134 ► **WheelGraph( *N* )** O

► **WheelGraph( *N*, *Radius* )** O

```

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]])

```



# Bibliography

# 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”.

## A

- A taxonomy of graphs, *4*
- AddEdges, 15, 35
- Adjacencies, 29, 35
- Adjacency, 29, 35
- AdjMatrix, 27, 35
- AGraph, 36
- AntennaGraph, 21, 36
- Attributes and properties of graphs, *27*

## B

- BackTrack, 36
- BackTrackBag, 36
- Basement, 36
- Binary operations, *23*
- BoxProduct, 23, 36
- BoxTimesProduct, 24, 36
- BullGraph, 20, 36

## C

- CayleyGraph, 36
- ChairGraph, 37
- Circulant, 37
- ClawGraph, 20, 37
- CliqueGraph, 15, 37
- CliqueNumber, 28, 37
- Cliques, 28, 37
- ComplementGraph, 23, 37
- CompleteBipartiteGraph, 18, 38
- CompleteGraph, 16, 38
- CompleteMultipartiteGraph, 18, 38
- CompletesOfGivenOrder, 30, 38
- Composition, 26, 38
- Coordinates, 39
- CopyGraph, 14, 39
- Core Operations, *33*
- Creating Graphs, *6*
- CuadraticRingGraph, 39
- Cube, 21, 39

- CubeGraph, 16, 39
- CycleGraph, 16, 40
- CylinderGraph, 40

## D

- DartGraph, 40
- DeclareQtifyProperty, 40
- Default Category, *11*
- Definition of graphs, *4*
- Diameter, 31, 40
- DiamondGraph, 20, 40
- DiscreteGraph, 16, 40
- DisjointUnion, 24, 41
- Distance, 30, 41
- DistanceGraph, 32, 41
- DistanceMatrix, 31, 41
- Distances, 31, 42
- Distances, *30*
- DistanceSet, 32, 42
- Dodecahedron, 22, 42
- DominoGraph, 42
- Draw, 42
- DumpObject, 42

## E

- Edges, 30, 43
- Excentricity, 31, 43
- Experimenting on graphs, *8*

## F

- Families, *16*
- FanGraph, 19, 43
- FishGraph, 43

## G

- GemGraph, 43
- Graph, 12, 43
- Graph Categories, *9*
- GraphByAdjacencies, 13, 44
- GraphByAdjMatrix, 12, 44

GraphByCompleteCover, 13, 44  
 GraphByRelation, 13, 44  
 GraphByWalks, 13, 45  
 GraphCategory, 11, 45  
 Graphs, 9, 45  
 GraphSum, 25, 45  
 GraphToRaw, 45  
 GraphUpdateFromRaw, 46  
 GroupGraph, 46

## H

HouseGraph, 20, 46

## I

Icosahedron, 22, 46  
 in, 11, 46  
 InducedSubgraph, 14, 46  
 Information about graphs, 29  
 InNeigh, 46  
 IntersectionGraph, 14, 46  
 IsBoolean, 47  
 IsCliqueGated, 47  
 IsCliqueHelly, 28, 47  
 IsComplete, 47  
 IsCompleteGraph, 28, 48  
 IsDiamondFree, 48  
 IsEdge, 48  
 IsIsomorphicGraph, 48  
 IsLoopless, 28, 48  
 IsoMorphism, 48  
 IsoMorphisms, 48  
 IsOriented, 28, 48  
 IsSimple, 29, 48  
 IsTournament, 48  
 IsTransitiveTournament, 48  
 IsUndirected, 28, 48

## J

JohnsonGraph, 17, 48  
 Join, 25, 49

## K

KiteGraph, 21, 49

## L

LineGraph, 22, 49  
 LooplessGraphs, 9, 49

## M

MaxDegree, 49  
 MinDegree, 49

Morphisms, 34

## N

NextIsoMorphism, 50  
 NextPropertyMorphism, 33, 50  
 NumberOfCliques, 50

## O

OctahedralGraph, 17, 50  
 Octahedron, 21, 50  
 Order, 27, 50  
 OrientedGraphs, 10, 50  
 OutNeigh, 51

## P

ParachuteGraph, 51  
 ParapluieGraph, 51  
 PathGraph, 16, 51  
 PawGraph, 20, 51  
 PowerGraph, 32, 51  
 Primitives, 12  
 PropertyMorphism, 33, 51  
 PropertyMorphisms, 33, 51

## Q

QtfyIsCliqueHelly, 28, 47  
 QtfyIsCompleteGraph, 28, 48  
 QtfyIsLoopless, 28, 48  
 QtfyIsOriented, 28, 48  
 QtfyIsSimple, 29, 51  
 QtfyIsUndirected, 28, 48  
 QuotientGraph, 23, 52

## R

Radius, 31, 52  
 RandomGraph, 18, 52  
 RemoveEdges, 15, 52  
 RemoveVertices, 14, 52  
 RGraph, 53  
 RingGraph, 53

## S

SetCoordinates, 53  
 SetDefaultGraphCategory, 11, 53  
 SimpleGraphs, 10, 54  
 Size, 27, 54  
 SnubDisphenoid, 54  
 SpikyGraph, 19, 54  
 SunGraph, 19, 54

## T

TargetGraphCategory, 11, 54  
Tetrahedron, 21, 54  
TimesProduct, 24, 54  
Transforming graphs, 8  
TrivialGraph, 19, 55

## U

Unary operations, 22  
UndirectedGraphs, 9, 55  
UnitsRingGraph, 55

Using YAGS, 3

## V

VertexDegree, 29, 55  
VertexDegrees, 30, 55  
VertexNames, 27, 55  
Vertices, 56

## W

WheelGraph, 18, 56