YAGS
Yet Another Graph System
GAP4 Package

Version 0.0.1

by

R. Mac Kinney Romero[1]

M. A. Pizaña[1]

R. Villarroel-Flores[2]

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

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

Partially supported by SEP-CONACyT, grant 183210.

October 14, 2015

Contents

1 Preface

1.1 Welcome to YAGS

Welcome to YAGS.
which is great and have a lot of work over the years.
Our motivation here was this and that.
Our Pourposes and Aim.
authors, contacts

1.2 Citing YAGS

Please cite us like this: ...

1.3 License and Copyright

GPL V3

1.4 More Information

web page, distribution list, github

2 Getting Started

2.1 What is YAGS?

blah, blah, blah

2.2 Installing YAGS

blah, blah, blah

2.3 Testing the Installation

blah, blah, blah

2.4 A Gentle Tutorial

blah, blah, blah

2.5 An Overview of the Manual

blah, blah, blah

2.6 Cheatsheet

blah, blah, blah

2.7 ——- Old Sections Bellow —-

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 the field of graph theory. 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

2.8 Using YAGS

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

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

Loading  YAGS 0.0.1  (Yet Another Graph System),
by  R. MacKinney, M.A. Pizana and R. Villarroel-Flores
rene@xamanek.izt.uam.mx, map@xamanek.izt.uam.mx, rvf0068@gmail.com

true
```

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

```
gap> ?PathGraph
Help: Showing ‘yags: PathGraph’

> PathGraph( <n> )                                                       F

Returns the path graph on <n> vertices.

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

2.9 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

$$\begin{tikzcd}
1\bullet & 2\bullet \arrow[d] \\
3\bullet \arrow[u] \arrow[ur] & 4\bullet
\end{tikzcd}$$

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

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

$$\begin{tikzcd}
\bullet \arrow[r, bend left] & \bullet \arrow[l, bend left] \arrow[r, phantom, "\equiv"] & \bullet \arrow[r, dash] & \bullet
\end{tikzcd}$$

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

$$\begin{tikzcd}
\bullet \arrow[loop right]
\end{tikzcd}$$

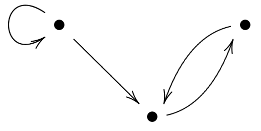
YAGS handles graphs that have arrows, edges and loops. Graphs that, for instance, have multiple arrows between vertices are not handled by YAGS.

$$\begin{tikzcd}
\bullet \arrow[r, bend left] \arrow[r] \arrow[r, bend right] & \bullet
\end{tikzcd}$$

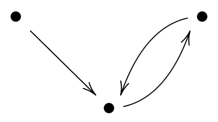
2.10 A taxonomy of graphs

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.

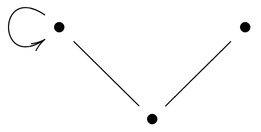
- *Graphs*: graphs with no particular property.



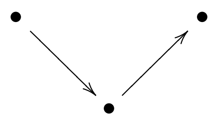
- *Loopless*: graphs with no loops.



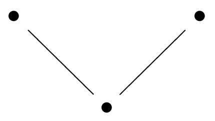
- *Undirected*: graphs with no arrows but only edges.



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



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



The following figure shows the relationships among categories.

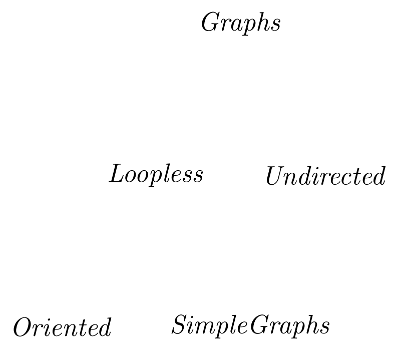


Figure 1: Graph Categories

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 2.11. 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 4.

2.11 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 2.9 the adjacency list would be

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

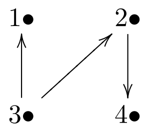
and the adjacency matrix

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

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

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

This will create a graph g that represents the graph in Section 2.9.



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 ], [  ] ] )
```

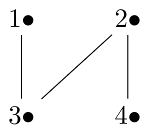
Creates the same graph. Note that we explicitly give the graph category as before. We also can use the command *AdjMatrix* as in

```
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

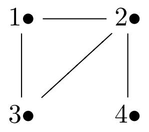


There are many functions to create graphs, some from existing graphs and some create interesting well known graphs.

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 ] ] )
```

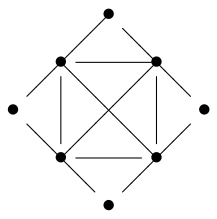
that yields the graph *h*



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 7.3.

3 Cliques

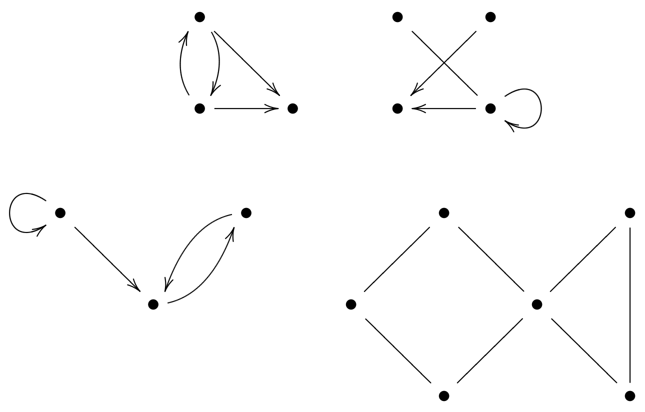
blah, blah, blah.

4 Categories

4.1 Graph Categories

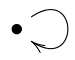
Declaration of: Graphs

Among them we can find:

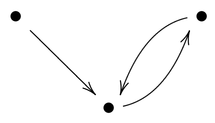


Declaration of: LooplessGraphs

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



Loopless graphs have no such arrows.

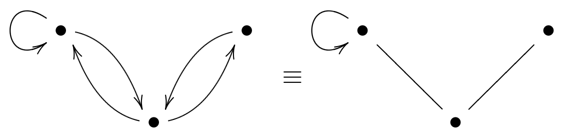


Declaration of: UndirectedGraphs

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

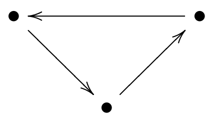
$$\begin{tikzcd}\bullet \arrow[r, bend left] & \bullet \arrow[l, bend left] & \equiv & \bullet \arrow[r, dash] & \bullet\end{tikzcd}$$

Undirected graphs have no arrows but only edges.

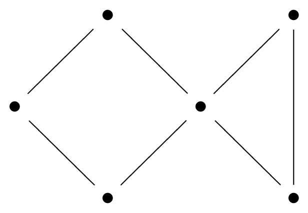


Declaration of: OrientedGraphs

Oriented graphs have no edges but only arrows.



Declaration of: SimpleGraphs



The following figure shows the relationships among categories.

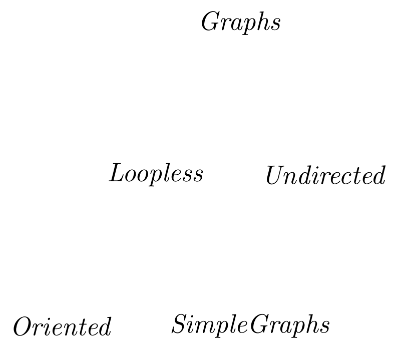
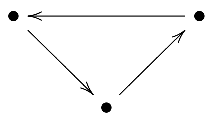
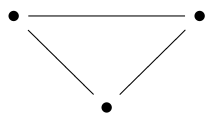


Figure 2: Graph Categories

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.

4.2 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.

Declaration of: SetDefaultGraphCategory

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

Declaration of: GraphCategory

Declaration of: TargetGraphCategory

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

Declaration of: in

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.

- CHK_METRIC *Metric*
- CHK_MONO *Mono*
- CHK_FULL *Full*
- CHK_EPI *Epi*
- CHK_CMPLT *Complete*
- CHK_ISO *Iso*

Additionally it must have at least one of the following.

- CHK_WEAK *Weak*
- CHK_MORPH *Morph*

These properties are detailed in the next section.

Declaration of: PropertyMorphism

Declaration of: PropertyMorphisms

Declaration of: NextPropertyMorphism

5.2 Morphisms

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

Metric A morphism is metric if the distance (see section 8) of any two vertices remains constant

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

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

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

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

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

Not yet implemented.

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

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

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

Iso An isomorphism is a bimorphism which is also complete.

Aditionally they must be one of the following

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

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

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

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

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

6 Backtrack

blah, blah, blah.

7 YAGS Functions by Topic

A complete list of all YAGS's functions by topic.

7.1 Most Common Functions

`AddEdges(` G, E `)`
: Returns a new graph obtained from G by adding the list of edges in E.

`Adjacency(` G, x `)`
: Returns the list of vertices in G which are adjacent to vertex x.

`AutGroupGraph(` G `)`
: Returns the automorphism group of graph G. A synonym is `AutomorphismGroup(` G `)`.

`BoxProduct(` G, H `);`
: Returns the BoxProduct (or cartesian product) of graphs G and H.

`BoxTimesProduct(` G, H `)`
: Returns the BoxTimesProduct (or strong product) of graphs G and H.

`Circulant(` n, *Jumps* `)`
: Returns minimal $(1, 2, \cdots, n)$-invariant graph where vertex 1 is adjacent to vertices in *Jumps*.

`CliqueGraph(` G `)`

`CliqueGraph(` G, *maxNumCli* `)`
: Returns the intersection graph of the (maximal) cliques of G; aborts if *maxNumCli* cliques are found.

`Cliques(` G `)`

`Cliques(` G, *maxNumCli* `)`
: Returns the list of (maximal) cliques of G; aborts if *maxNumCli* cliques are found.

`ComplementGraph(` G `)`
: Returns the new graph H such that $V(H) = V(G)$ and $xy \in E(H) \iff xy \notin E(G)$.

`CompleteGraph(` n `)`
: Returns the graph on n vertices having all possible edges present.

`CompleteMultipartiteGraph(` *n1*, *n2* `[,` *n3* `...] )`
: Returns the graph with $r \geq 2$ parts of orders *n1*, *n2*, ... such that each vertex is adjacent exactly to all the vertices in the other parts not containing itself.

`ConnectedComponents(` G `)`
: Returns the equivalece partition of $V(G)$ corresponding to the equivalence relation **reachable** in G.

`CycleGraph(` n `)`
: Returns the cyclic graph on n vertices.

`Diameter(` G `)`
: Returns the maximum among the distances between pairs of vertices of G.

`DiscreteGraph(` *n* `)`
: Returns the graph on n vertices with no edges.

`DisjointUnion(` *G*`,` *H* `)`
: Returns the disjoint union of two graphs G and H.

`Distance(` *G*`,` *x*`,` *y* `)`
: Returns the length of a minimal path connecting x to y in G.

`Draw(` *G* `)`
: Draws the graph G on a new window.

`Edges(` *G* `)`
: Returns the list of edges of graph G.

`GraphAttributeStatistics(` *OrderList*`,` *ProbList*`,` *Attribute* `)`
: Returns statistics for graph attribute *Attribute*.

`GraphByAdjacencies(` *AdjList* `)`
: Returns a new graph having *AdjList* as its list of adjacencies.

`GraphByAdjMatrix(` *Mat* `)`
: Returns a new graph created from an adjacency matrix *Mat*.

`GraphByCompleteCover(` *Cover* `)`
: Returns the graph where the elements of *Cover* are (the vertex sets of) complete subgraphs.

`GraphByEdges(` *L* `)`
: Returns the minimal graph such that the pairs in L are edges.

`GraphByRelation(` *V*`,` *Rel* `)`

`GraphByRelation(` *n*`,` *Rel* `)`
: Returns a new graph G where $xy \in E(G)$ iff *Rel(x,y)=true*.

`GraphByWalks(` *Walk1*`,` *Walk2*`,...)`
: Returns the minimal graph such that *Walk1*, *Walk2*, etc are Walks.

`GraphSum(` *G*`,` *L* `)`
: Returns the lexicographic sum of a list of graphs L over a graph G.

`InducedSubgraph(` *G*`,` *V* `)`
: Returns the subgraph of graph G induced by the vertex set V.

`InNeigh(` *G*`,` *x* `)`
: Returns the list of in-neighbors of x in G.

`IntersectionGraph(` *L* `)`
: Returns the graph G where $V(G) = L$ and $XY \in E(G) \iff X \cap Y \neq \varnothing$.

`IsEdge(` *G* `,` *x*`,` *y* `)`

`IsEdge(` *G* `, [`*x*`,`*y*`] )`
: Returns `true` if `[`x`,`y`]` is an edge of G.

`IsIsomorphicGraph(` *G*`,` *H* `)`
: Returns `true` when G is isomorphic to H and `false` otherwise.

`Join(` *G*`,` *H* `)`
: Returns the disjoint union of G and H with all the possible edges between G and H added.

`LineGraph(` *G* `)`
: Returns the intersection graph of the edges of G.

`Link(` *G*`,` *x* `)`
: Returns the subgraph induced in G by the neighbors of x.

`MaxDegree(` *G* `)`
: Returns the maximum degree in graph G.

`Order(`*G*`)`
: Returns the number of vertices, of graph G.

`PathGraph(` *n* `)`
: Returns the path graph on n vertices.

`QuotientGraph(` *G*`,` *Part* `)`

`QuotientGraph(` *G*`,` *L1*`,` *L2* `)`
: Returns the quotient graph of graph G given a vertex partition *Part*, by identifying any two vertices in the same part.

`RandomGraph(` *n*`,` *p* `)`

`RandomGraph(` *n* `)`
: Returns a random graph of order n with edge probability p (a rational in $[0,1]$).

`RemoveEdges(` *G*`,` *E* `)`
: Returns a new graph created from graph G by removing the edges in list E.

`SetDefaultGraphCategory(` *Catgy* `)`
: Sets the default graph category to *Catgy*.

`Size(`*G*`)`
: Returns the number of edges of graph G.

`TimesProduct(` *G*`,` *H* `)`
: Returns the times product (tensor product) $G \times H$ of two graphs G and H.

`TrivialGraph`
: The one vertex graph.

`VertexDegree(` *G*`,` *x* `)`
: Returns the degree of vertex x in Graph G.

`VertexNames(`*G*`)`
: Returns the list of names of the vertices of G.

`WheelGraph(` *n* `)`

`WheelGraph(` *n*`,` *r* `)`
: This is the cone of an n-cycle; when present r is the radius of the wheel.

7.2 Drawing

`Coordinates(` *G* `)`
: Returns the list of coordinates of the vertices of G if they exist; fail otherwise.

`Draw(` *G* `)`
: Draws the graph G on a new window.

`GraphToRaw(` *FileName*`,` *G* `)`
: Writes the graph G in raw format to the file *FileName*.

`GraphUpdateFromRaw(` *FileName*`,` *G* `)`
: Updates the coordinates of G from a file *FileName* in raw format.

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

7.3 Constructing Graphs

`AddEdges(` *G*`,` *E* `)`

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

`AddVerticesByAdjacencies(` *G*`,` *NewAdjList* `)`

Returns a new graph obtained from G by adding some vertices with adjacencies described by *NewAdjList*.

`Graph(` *Rec* `)`

Returns a new graph created from the information in record *Rec*.

`GraphByAdjacencies(` *AdjList* `)`

Returns a new graph having *AdjList* as its list of adjacencies.

`GraphByAdjMatrix(` *Mat* `)`

Returns a new graph created from an adjacency matrix *Mat*.

`GraphByCompleteCover(` *Cover* `)`

Returns the graph where the elements of *Cover* are (the vertex sets of) complete subgraphs.

`GraphByEdges(` *L* `)`

Returns the minimal graph such that the pairs in L are edges.

`GraphByRelation(` *V*`,` *Rel* `)`

`GraphByRelation(` *n*`,` *Rel* `)`

Returns a new graph G where $xy \in E(G)$ iff *Rel(x,y)=true*.

`GraphByWalks(` *Walk1*`,` *Walk2*`,...)`

Returns the minimal graph such that *Walk1*, *Walk2*, etc are Walks.

`IntersectionGraph(` *L* `)`

Returns the graph G where $V(G) = L$ and $XY \in E(G) \iff X \cap Y \neq \varnothing$.

`RandomGraph(` *n*`,` *p* `)`

`RandomGraph(` *n* `)`

Returns a random graph of order n with edge probability p (a rational in $[0,1]$).

`RemoveEdges(` *G*`,` *E* `)`

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

`RemoveVertices(` *G*`,` *V* `)`

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

7.4 Families of Graphs

`AGraph`

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

`AntennaGraph`

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

`BullGraph`

A triangle with two pendant vertices (horns).

`ChairGraph`

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

`Circulant(` *n*`,` *Jumps* `)`

Returns minimal $(1, 2, \cdots, n)$-invariant graph where vertex 1 is adjacent to vertices in *Jumps*.

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

`ClockworkGraph(` *NNFSList* `)`

`ClockworkGraph(` *NNFSList*`,` *rank* `)`

`ClockworkGraph(` *NNFSList*`,` *Perm* `)`

`ClockworkGraph(` *NNFSList*`,` *rank*`,` *Perm* `)`
: Returns the clockwork graph specified by its parameters.

`CompleteBipartiteGraph(` *n*`,` *m* `)`
: Returns the minimal graph such that all vertices in $\{1 \cdot\cdot n\}$ are adjacent to all in $\{n+1 \cdot\cdot n+m\}$.

`CompleteGraph(` *n* `)`
: Returns the graph on n vertices having all possible edges present.

`CompleteMultipartiteGraph(` *n1*`,` *n2* `[,` *n3* `...] )`
: Returns the graph with $r \geq 2$ parts of orders *n1*, *n2*, ... such that each vertex is adjacent exactly to all the vertices in the other parts not containing itself.

`Cube`
: The 1-skeleton of Plato's cube.

`CubeGraph(` *n* `)`
: Returns the underlying graph of the n-hypercube.

`CycleGraph(` *n* `)`
: Returns the cyclic graph on n vertices.

`CylinderGraph(` *b*`,` *h* `)`
: Returns graph on $b(h+1)$ vertices which is a $\{4,6\}$-regular triangulation of the cylinder.

`DartGraph`
: A diamond with a pendant vertex and maximum degree 4.

`DiamondGraph`
: The graph on 4 vertices and 5 edges.

`DiscreteGraph(` *n* `)`
: Returns the graph on n vertices with no edges.

`Dodecahedron`
: The 1-skeleton of Plato's Dodecahedron.

`DominoGraph`
: Two squares glued by an edge.

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

`FishGraph`
: A square and a triangle glued by a vertex.

`GemGraph`
: The 3-Fan graph.

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

`Icosahedron`
: The 1-skeleton of Plato's icosahedron.

`JohnsonGraph(` *n*`,` *r* `)`
: Returns a new graph G where *V(G)* is the set of r-subsets of $\{1, 2 \ldots n\}$, two of them being adjacent iff their intersection contains exactly r-1 elements.

`KiteGraph`
: A diamond with a pendant vertex and maximum degree 3.

`OctahedralGraph(` *n* `)`
: Returns the *(2n-2)*-regular graph on $2n$ vertices.

`Octahedron`
: The 1-skeleton of Plato's octahedron.

`ParachuteGraph`
: Returns the suspension of a 4-path with a pendant vertex attached to the south pole.

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

`PathGraph(` *n* `)`
: Returns the path graph on n vertices.

`PawGraph`
: A triangle with a pendant vertex.

`PetersenGraph`
: The 3-regular graph on 10 vertices having girth 5.

`RandomCirculant(` *n* `)`

`RandomCirculant(` *n*`,` *k*`)`

`RandomCirculant(` *n*`,` *p*`)`
: Returns a circulant on n vertices with its *jumps* selected randomly.

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

`SnubDisphenoid`
: The 1-skeleton of the 84th Johnson solid.

`SpikyGraph(` *n* `)`
: Returns a complete on n vertices, with an additional complete on n vertices glued to each of its n-1 (n-1)-dimensional faces.

`SunGraph(` *n* `)`
: Returns a complete graph on n vertices with a zigzaging corona of $2n$ vertices glued to a n-cycle of the complete graph.

`Tetrahedron`
: The 1-skeleton of Plato's tetrahedron.

`TorusGraph(` *n*`,` *m* `)`
: Returns (the underlying graph of) a triangulation of the torus on $n \cdot m$ vertices.

`TreeGraph(` *arity*`,` *depth* `)`

`TreeGraph(` *ArityList* `)`
: Returns the tree, the connected cycle-free graph, specified by it parameters.

`TrivialGraph`
: The one vertex graph.

`WheelGraph(` *n* `)`

`WheelGraph(` *n*`,` *r* `)`
: This is the cone of an n-cycle; when present r is the radius of the wheel.

7.5 Small Graphs

`ConnectedGraphsOfGivenOrder(` *n* `)`
: Returns the list of all connected graphs of order n (upto isomorphism).

`Graph6ToGraph(` *String* `)`
: Returns the graph represented by *String* which is encoded using Brendan McKay's graph6 format.

`GraphsOfGivenOrder(` *n* `)`
: Returns the list of all graphs of order n (upto isomorphism).

`ImportGraph6(` *Filename* `)`
: Returns the list of graphs represented in *Filename* which are encoded using Brendan McKay's graph6 format.

`HararyToMcKay(` *Spec* `)`
: Returns the McKay's *index* of a Harary's graph specification *Spec*.

`McKayToHarary(` *index* `)`
: Returns the Harary's graph specification of a McKay's *index*.

7.6 Attributes and Properties

`Adjacencies(` *G* `)`
: Returns the list of adjacencies of G: The neighbors of vertex x are listed in position x of that list.

`Adjacency(` *G*`,` *x* `)`
: Returns the list of vertices in G which are adjacent to vertex x.

`AdjMatrix(` *G* `)`
: Returns the Adjacency Matrix of G.

`AutGroupGraph(` *G* `)`
: Returns the automorphism group of graph G. A synonym is `AutomorphismGroup(` *G* `)`.

`ConnectedComponents(` *G* `)`
: Returns the equivalece partition of $V(G)$ corresponding to the equivalence relation **reachable** in G.

`Diameter(` *G* `)`
: Returns the maximum among the distances between pairs of vertices of G.

`Distance(` *G*`,` *x*`,` *y* `)`
: Returns the length of a minimal path connecting x to y in G.

`DistanceMatrix(` *G* `)`
: Returns an $n \times n$ matrix D, where $D[x][y]$ is the distance between x and y in G.

`DistanceSet(` *G*`,` *A*`,` *B* `)`
: Returns the set of distances between pairs of vertices in $A \times B$.

`Distances(` *G*`,` *A*`,` *B* `)`
: Returns the list of distances between pairs of vertices in $A \times B$.

`DominatedVertices(` *G* `)`
: Returns the set of dominated vertices of G.

`Eccentricity(` *G*`,` *x* `)`
: Returns the distance from a vertex x in graph G to its most distant vertex in G.

`Edges(` *G* `)`
: Returns the list of edges of graph G.

`Girth(` *G* `)`
: Returns the length of the minimum induced cycle in *G*.

`GraphAttributeStatistics(` *OrderList*, *ProbList*, *Attribute* `)`
: Returns statistics for graph attribute *Attribute*.

`IsDiamondFree(` *G* `)`
: Returns `true` if *G* is free from induced diamonds, `false` otherwise.

`IsEdge(` *G* , *x*, *y* `)`

`IsEdge(` *G* , `[`*x*,*y*`]` `)`
: Returns `true` if `[`*x*,*y*`]` is an edge of *G*.

`IsLoopless(`*G*`)`
: Returns `true` when *G* is isomorphic to *H* and `false` otherwise.

`IsOriented(` *G* `)`
: Returns `true` if whenever xy is an edge (arrow) of *G*, yx is not.

`IsSimple(` *G* `)`
: Returns `true` if *G* contains no loops and no arrows.

`IsUndirected(`*G*`)`
: Returns `true` if whenever xy is an edge (arrow) of *G*, yx is also an edge of *G*.

`Link(` *G*, *x* `)`
: Returns the subgraph induced in *G* by the neighbors of *x*.

`Links(` *G* `)`
: Returns the list of subgraphs of *G* induced by the neighbors of each vertex of *G*.

`MaxDegree(` *G* `)`
: Returns the maximum degree in graph *G*.

`MinDegree(` *G* `)`
: Returns the minimum degree in graph *G*.

`NumberOfConnectedComponents(` *G* `)`
: Returns the number of connected components of *G*.

`Order(`*G*`)`
: Returns the number of vertices, of graph *G*.

`Radius(` *G* `)`
: Returns the minimal eccentricity among the vertices of graph *G*.

`Size(`*G*`)`
: Returns the number of edges of graph *G*.

`SpanningForest(` *G* `)`
: Returns a spanning forest of *G*.

`SpanningForestEdges(` *G* `)`
: Returns the edges of a spanning forest of *G*.

`VertexDegree(` *G*, *x* `)`
: Returns the degree of vertex *x* in Graph *G*.

`VertexDegrees(` *G* `)`
: Returns the list of degrees of the vertices in graph *G*.

`VertexNames(`*G*`)`
: Returns the list of names of the vertices of *G*.

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

7.7 Unary Operators

`ComplementGraph(` *G* `)`
: Returns the new graph H such that $V(H) = V(G)$ and $xy \in E(H) \iff xy \notin E(G)$.

`CompletelyParedGraph(` *G* `)`
: Returns the graph obtained from G by iteratively removing all dominated vertices.

`Cone(` *G* `)`
: Returns a new graph obtained from G by adding a new vertex which is adjacent to all vertices of G.

`CliqueGraph(` *G* `)`
`CliqueGraph(` *G*`,` *maxNumCli* `)`
: Returns the intersection graph of the (maximal) cliques of G; aborts if *maxNumCli* cliques are found.

`DistanceGraph(` *G*`,` *Dist* `)`
: Returns a new graph where two vertices are adjacent iff the distance between them belongd to *Dist*.

`InducedSubgraph(` *G*`,` *V* `)`
: Returns the subgraph of graph G induced by the vertex set V.

`LineGraph(` *G* `)`
: Returns the intersection graph of the edges of G.

`ParedGraph(` *G* `)`
: Returns the induced subgraph obtained from G by removing its dominated vertices.

`PowerGraph(` *G*`,` *exp* `)`
: Returns a new graph where two vertices are neighbors iff their distance in G is less than or equal to *exp*.

`QuotientGraph(` *G*`,` *Part* `)`
`QuotientGraph(` *G*`,` *L1*`,` *L2* `)`
: Returns the quotient graph of graph G given a vertex partition *Part*, by identifying any two vertices in the same part.

`Suspension(` *G* `)`
: Returns the graph obtained from G by adding two new vertices which are adjacent to every vertex of G but not to each other.

7.8 Binary Operators

`BoxProduct(` *G*`,` *H* `);`
: Returns the BoxProduct (or cartesian product) of graphs G and H.

`BoxTimesProduct(` *G*`,` *H* `)`
: Returns the BoxTimesProduct (or strong product) of graphs G and H.

`Composition(` *G*`,` *H* `)`
: Returns the composition $G[H]$ of two graphs G and H.

`DisjointUnion(` *G*`,` *H* `)`
: Returns the disjoint union of two graphs G and H.

`GraphSum(` *G*`,` *L* `)`
: Returns the lexicographic sum of a list of graphs L over a graph G.

`Join(` *G*`,` *H* `)`
: Returns the disjoint union of G and H with all the possible edges between G and H added.

`TimesProduct(` *G*`,` *H* `)`
: Returns the times product (tensor product) $G \times H$ of two graphs G and H.

7.9 Cliques

`Basement(` G`,` KnG`,` x `)`

`Basement(` G`,` KnG`,` V `)`

Returns the basement of vertex x (vertex set V) of the iterated clique graph KnG with respect to G.

`CliqueGraph(` G `)`

`CliqueGraph(` G`,` *maxNumCli* `)`

Returns the intersection graph of the (maximal) cliques of G; aborts if *maxNumCli* cliques are found.

`CliqueNumber(` G `)`

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

`Cliques(` G `)`

`Cliques(` G`,` *maxNumCli* `)`

Returns the list of (maximal) cliques of G; aborts if *maxNumCli* cliques are found.

`CompletesOfGivenOrder(` G`,` *Ord* `)`

Returns the list of vertex sets of all complete subgraphs of order *Ord* of G.

`IsCliqueGated(` G `)`

Returns `true` if G is a clique gated graph.

`IsCliqueHelly(` G `)`

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

`IsComplete(` G`,` L `)`

Returns `true` if L induces a complete subgraph of G.

`IsCompleteGraph(`G`)`

Returns `true` if graph G is a complete graph, `false` otherwise.

`NumberOfCliques(` G `)`

`NumberOfCliques(` G`,` *maxNumCli* `)`

Returns the number of (maximal) cliques of G.

7.10 Morphisms and Isomorphisms

`IsIsomorphicGraph(` G`,` H `)`

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

`IsoMorphism(` G`,` H `)`

Returns one isomorphism from G to H; `fail` if there is none.

`IsoMorphisms(` G`,` H `)`

Returns the list of all isomorphism from G to H.

`NextIsoMorphism(` G`,` H`,` F `)`

Returns the next isomorphism (after F) from G to H.

`NextPropertyMorphism(` G`,` H`,` F`,` *PropList* `)`

Returns the next morphism (after F) from G to H satisfying the list of properties *PropList*.

`PropertyMorphism(` G`,` H`,` *PropList* `)`

Returns the first morphism from G to H satisfying the list of properties *PropList*.

`PropertyMorphisms(` G`,` H`,` *PropList* `)`

Returns all morphisms from G to H satisfying the list of properties *PropList*.

7.11 Graph Categories

`CopyGraph(` *G* `)`
: Returns a fresh copy of G. Useful to change the category of a graph.

`GraphCategory( [` *G*`, ... ] );`
: For internal use. Returns the minimal common category to a list of graphs.

`Graphs()`
: The category of all graphs that can be represented in YAGS.

`in(`*G*`,`*Catgy*`)`
: Returns `true` if graph G belongs to category *Catgy* and `false` otherwise.

`LooplessGraphs()`
: The category of all graph that may contain arrows and edges but no loops.

`OrientedGraphs()`
: The category of all graphs that may contain arrows but no edges or loops.

`SetDefaultGraphCategory(` *Catgy* `)`
: Sets the default graph category to *Catgy*.

`SimpleGraphs()`
: The category of all graphs which may contain edges but no arrows or loops.

`TargetGraphCategory( [` *G*`, ... ] );`
: For internal use. Within YAGS methods, returns the graph category to which the new graph will belong.

`UndirectedGraphs()`
: The category of all graphs that may contain loops and edges but no arrows.

7.12 Digraphs

`InNeigh(` *G*`,` *x* `)`
: Returns the list of in-neighbors of x in G.

`IsTournament(` *G* `)`
: Returns `true` if G contains no loops or edges but only arrows and it is maximal w.r.t. this property.

`IsTransitiveTournament(` *G* `)`
: Returns `true` if G is a Tournament and whenever xy and yz are arrows, then xz is an arrow too.

`OutNeigh(` *G*`,` *x* `)`
: Returns the list of out-neighbors of x in G.

7.13 Groups and Rings

`CayleyGraph(` *Grp* `)`

`CayleyGraph(` *Grp*`,` *Elms* `)`
: Returns the CayleyGraph of group *Grp*.

`Circulant(` *n*`,` *Jumps* `)`
: Returns minimal $(1,2,\cdots,n)$-invariant graph where vertex 1 is adjacent to vertices in *Jumps*.

`CuadraticRingGraph(` *Rng* `)`
: Returns a graph H whose vertices are the elements of the ring *Rng* and $xy \in E(H) \iff x+z^2=y$ for some z in *Rng*.

`GroupGraph(` *G*`,` *Grp* `)`

`GroupGraph(` *G*`,` *Grp*`,` *Act* `)`

Returns the minimal *Grp*-invariant (under the action *Act*) graph containing *G*.

`RingGraph(` *Rng*`,` *Elms* `)`

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*.

`UnitsRingGraph(` *Rng* `)`

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*.

7.14 Backtrack

`BackTrack(` *L*`,` *Opts*`,` *Chk*`,` *Done*`,` *Extra* `)`

Returns the next solution (after *L*) to a backtracking combinatorial problem specified by its parameters.

`BackTrackBag(` *Opts*`,` *Chk*`,` *Done*`,` *Extra* `)`

Returns the list of all solutions to a backtracking combinatorial problem specified by its parameters.

7.15 Miscellaneous

`DumpObject(` *Obj* `)`

For internal use. Dumps all information available for object *Obj*.

`EasyExec(` *Dir*`,` *ProgName*`,` *InString* `)`

`EasyExec(` *ProgName*`,` *InString* `)`

Calls the external program *ProgName* with input string *InString*; returns the output string.

`IsBoolean(` *Obj* `)`

Returns `true` if object *Obj* is `true` or `false` and `false` otherwise.

`QtfyIsSimple(` *G* `)`

For internal use. Returns how far is graph *G* from being simple.

`RandomlyPermuted(` *Obj* `)`

Returns a copy of *Obj* with the order of its elements permuted randomly.

`RandomPermutation(` *n* `)`

Returns a random permutation of the list `[1,2, ...,` *n*`]`.

`RandomSubset(` *Set* `)`

`RandomSubset(` *Set*`,` *k* `)`

`RandomSubset(` *Set*`,` *p* `)`

Returns a random subset of the set *Set*. It also works for lists though.

`TimeInSeconds()`

Returns the time in seconds since 1970-01-01 00:00:00 UTC as an integer.

`UFFind(` *UFS*`,` *x* `)`

For internal use. Implements the *find* operation on the *union-find structure*.

`UFUnite(` *UFS*`,` *x*`,` *y* `)`

For internal use. Implements the *unite* operation on the *union-find structure*.

`YAGSExec(` *ProgName*`,` *InString* `)`

For internal use. Calls external program *ProgName* located in directory '*YAGSDir*/bin/' feeding it with *InString* as input and returning the output of the external program as a string.

7.16 Undocumented

`DeclareQtfyProperty(` *Name*`,` *Filter* `)`
: For internal use. Declare a quantifiable property.

`DumpObject(` *Obj* `)`
: For internal use. Dumps all information available for object *Obj*.

`EasyExec(` *Dir*`,` *ProgName*`,` *InString* `)`

`EasyExec(` *ProgName*`,` *InString* `)`
: Calls the external program *ProgName* with input string *InString*; returns the output string.

`GraphToRaw(` *FileName*`,` *G* `)`
: Writes the graph *G* in raw format to the file *FileName*.

`GraphUpdateFromRaw(` *FileName*`,` *G* `)`
: Updates the coordinates of *G* from a file *FileName* in raw format.

`QtfyIsSimple(` *G* `)`
: For internal use. Returns how far is graph *G* from being simple.

`YAGSExec(` *ProgName*`,` *InString* `)`
: For internal use. Calls external program *ProgName* located in directory '*YAGSDir*/bin/' feeding it with *InString* as input and returning the output of the external program as a string.

`YAGSInfo`
: For internal use. A global record where much YAGS-related information is stored.

8 YAGS Functions Reference

This chapter contains a complete list of all YAGS's functions, with definitions, in alphabetical order.

1 ▶ `AddEdges(` *G*`,` *E* `)` O

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

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

2 ▶ `AddVerticesByAdjacencies(` *G*`,` *NewAdjList* `)` O

Returns a new graph created from graph *G* by adding as many new vertices as `Length(`*NewAdjList*`)`. Each entry in *NewAdjList* is also a list: the list of neighbors of the corresponding new vertex.

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

3 ▶ `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 ] ]
```

4 ▶ `Adjacency(` *G*`,` *x* `)` O

Returns the adjacency list of vertex *x* 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 ]
```

5 ► `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 ] ]
```

6 ► `AGraph` V

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

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

7 ► `AntennaGraph` V

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

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

8 ► `AutGroupGraph(` *G* `)` A

Returns the group of automorphisms of the graph *G*. There is also a synonym for this attribute which is `AutomorphismGroup(` *G* `)`.

```
gap> AutGroupGraph(Icosahedron);
Group([ (1,3,2,10,9,12,8,7,5,4)(6,11), (1,7,9)(2,4,12)(3,11,10)(5,8,6) ])
gap> AutomorphismGroup(Icosahedron);
Group([ (1,3,2,10,9,12,8,7,5,4)(6,11), (1,7,9)(2,4,12)(3,11,10)(5,8,6) ])
```

9 ► `BackTrack(` *L*, *Opts*, *Chk*, *Done*, *Extra* `)` O

Generic, user-customizable backtracking algorithm.

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

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

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

Opts`:=function(L,extra)` must return the list of continuation options (childs) one has after some partial solution (node) *L* has been reached within the decision tree (*Opts* may use the extra data *Extra* as needed).

Each of the values in the list returned by *Opts*`(L,extra)` will be tried as possible continuations of the partial solution *L*. If *Opts*`(L,extra)` always returns the same list, you can put that list in place of the parameter *Opts*.

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

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

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

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

10 ► `BackTrackBag(` *Opts*, *Chk*, *Done*, *Extra* `)` O

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

The following example computes all derrangements of order 4.

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

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

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

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

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

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

12 ▶ `BoxProduct(` G, H `)` O

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

The box product is calculated as follows:

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

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

13 ▶ `BoxTimesProduct(` G, H `)` O

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

The boxtimes product is calculated as follows:

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

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

14 ► `BullGraph` V

A triangle with two pendant vertices (horns).

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

15 ► `CayleyGraph(` *Grp*, *Elms* `)` O
► `CayleyGraph(` *Grp* `)` O

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

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

16 ► `ChairGraph` V

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

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

17 ► `Circulant(` *n*, *Jumps* `)` O

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

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

18 ▶ `ClawGraph` V

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

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

19 ▶ `CliqueGraph(` *G* `)` A
▶ `CliqueGraph(` *G*, *maxNumCli* `)` O

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

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

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

20 ▶ `CliqueNumber(` *G* `)` A

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

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

21 ▶ `Cliques(` *G* `)` A
▶ `Cliques(` *G*, *maxNumCli* `)` O

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

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

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

22 ▶ `ClockworkGraph(` *NNFSList* `)` O
▶ `ClockworkGraph(` *NNFSList*, *rank* `)` O
▶ `ClockworkGraph(` *NNFSList*, *Perm* `)` O
▶ `ClockworkGraph(` *NNFSList*, *rank*, *Perm* `)` O

Returns the clockwork graph [LN02,LNP04] specified by its parameters. A clockwork graph consists of two parts: the crown and the core, both of them are *cyclically segmented*. When not specified, the *rank* is assumed to be 2 and the *return permutation*, *Perm*, is assumed to be trivial, let us assume this is our case. Consider the following examples:

```
gap> ClockworkGraph([[0],[0],[0],[0]]);
Graph( Category := SimpleGraphs, Order := 12, Size := 28, Adjacencies :=
[ [ 2, 3, 4, 10, 12 ], [ 1, 3, 5, 11, 12 ], [ 1, 2, 4, 5 ], [ 1, 3, 5, 6, 7 ],
  [ 2, 3, 4, 6, 8 ], [ 4, 5, 7, 8 ], [ 4, 6, 8, 9, 10 ], [ 5, 6, 7, 9, 11 ],
  [ 7, 8, 10, 11 ], [ 1, 7, 9, 11, 12 ], [ 2, 8, 9, 10, 12 ], [ 1, 2, 10, 11 ] ] )
gap> ClockworkGraph([[1],[1],[1],[1]]);
Graph( Category := SimpleGraphs, Order := 12, Size := 32, Adjacencies :=
[ [ 2, 3, 4, 10, 12 ], [ 1, 3, 5, 11, 12 ], [ 1, 2, 4, 5, 6, 12 ], [ 1, 3, 5, 6, 7 ],
  [ 2, 3, 4, 6, 8 ], [ 3, 4, 5, 7, 8, 9 ], [ 4, 6, 8, 9, 10 ], [ 5, 6, 7, 9, 11 ],
  [ 6, 7, 8, 10, 11, 12 ], [ 1, 7, 9, 11, 12 ], [ 2, 8, 9, 10, 12 ],
  [ 1, 2, 3, 9, 10, 11 ] ] )
```

In both cases, the crown is the subgraph induced by the vertices $\{1,2,4,5,7,8,10,11\}$ and the core is induced by $\{3,6,9,12\}$. Also in both cases the cyclic segmentations (partitions) of the crown and the core are $\{\{1,2\},\{4,5\},\{7,8\},\{10,11\}\}$ and $\{\{3\},\{6\},\{9\},\{12\}\}$ respectively. The number of segmentes s is specified by *s*`:=Length(`*NNFSList*`)` which is 4 in these cases. The crown is isomorphic to `BoxProduct(CycleGraph(`*s*`),Completegraph(`*rank*`))`: All the crown segments are complete subgraphs and the vertices of cyclically consecutive segments are joined by a perfect matching. The adjacencies between crown and core vertices are simple to describe: Cyclically intercalate crown and core segments, making each core vertex adjacent to the vertices in the previous and the following crown segments. Hence in our examples vertex 3 is adjacent to vertices 1 and 2 (previous segment), but also 4 and 5 (following segment). Note that since the segmentations and intercalations are *cyclic*, we have that vertex 12 is adjacent to 10 and 11, but also to 1 and 2. Finally the edges between core vertices are as follows: first each core segment is a complete subgraph; the vertices within each core segment are linearly ordered and for vertex number t in segment number s there is a non-negative integer *NNFSList*`[s][t]` which specifies, the *Number of Neighbors in the Following core Segment* for that vertex (hence the name *NNFSList*) (Since the vertices in core segments are linearly ordered, it is enough to specify the *number* of neighbors in the following segment and the *first* ones of those are selected as the neighbors). Hence in our two examples above, each core segment consists of exaclty one vertex. In the first example each core segment is adjacent to no vertex in the following segment (e.g. 3 is not adjacent to 6) but in the second one, each core segment is adjacent to exactly one vertex in the following segment (e.g. 3 is adjacent to 6).

A more complicated example should be now mostly self-explanatory:

```
gap> ClockworkGraph([[2],[0,1,3],[0,1,1],[1]]);
Graph( Category := SimpleGraphs, Order := 16, Size := 59, Adjacencies :=
[ [ 2, 3, 4, 14, 16 ], [ 1, 3, 5, 15, 16 ], [ 1, 2, 4, 5, 6, 7, 16 ],
  [ 1, 3, 5, 6, 7, 8, 9 ], [ 2, 3, 4, 6, 7, 8, 10 ], [ 3, 4, 5, 7, 8, 9, 10 ],
  [ 3, 4, 5, 6, 8, 9, 10, 11 ], [ 4, 5, 6, 7, 9, 10, 11, 12, 13 ],
  [ 4, 6, 7, 8, 10, 11, 12, 13, 14 ], [ 5, 6, 7, 8, 9, 11, 12, 13, 15 ],
  [ 7, 8, 9, 10, 12, 13, 14, 15 ], [ 8, 9, 10, 11, 13, 14, 15, 16 ],
  [ 8, 9, 10, 11, 12, 14, 15, 16 ], [ 1, 9, 11, 12, 13, 15, 16 ],
  [ 2, 10, 11, 12, 13, 14, 16 ], [ 1, 2, 3, 12, 13, 14, 15 ] ] )
```

The crown and core segmentations are $\{\{1,2\},\{4,5\},\{9,10\},\{14,15\}\}$ and $\{\{3\},\{6,7,8\},\{11,12,13\},\{16\}\}$ respectively and the adjacencies specified by the *NNFSList* are: 3 is adjacent to 6 and 7; 6 is adjacent to none (in the following core segment); 7 is adjacent to 11; 8 to 11, 12 and 13; 11 to none; 12 to 16; 13 to 16 and 16 to 3.

When *rank* and/or *Perm* are specified, they have the following effects: *rank* (which must be at least 2) is the number of vertices in each crown segment, and *Perm* (which must belong to SymmetricGroup(*rank*)) specifies the perfect matching joining the vertices in the last crown segment with the vertices in the first crown segment: The k-th vertex in the last crown segment $k \in \{1, 2, \ldots, rank\}$ is made adjacent to the $Perm(k)$-th vertex of the first crown segment.

A number of requisites are put forward in the literature for a graph to be a clockwork graph but this operation does not enforce those conditions, on the contrary, it tries to make sense of the data provided as much as possible. For instance *NNFSList*`:=[[2],[2],[2],[2]]` would be inconsistent since there are not enough vertices in each core segment to provide for the required 2 neighbors. However, the result is just the same as with *NNFSList*`:=[[1],[1],[1],[1]]`. The requisites that are mandatory are exactly these: the *rank* must be at least 2, *Perm* must belong to SymmetricGroup(*rank*), *NNFSList* must be a list of lists of non-negative integers, and the number of segments (= Length(*NNFSList*)) must be at least 3. A call to ClockworkGraph which fails to conform to these requisites will produce an error.

Clockwork graphs have been very useful in constructing examples and counter-examples in clique graph theory. In particular, they have been used to construct examples of clique-periodic graphs of all possible periods [Esc73], clique-divergent graphs of linear and polynomial growth rate [LN97,LN02], clique-convergent graphs whose period is not invariant under removal of dominated vertices [FNP04], clique-convergent graphs which become clique-divergent by just gluing a 4-cycle to a vertex [FLNP13], rank-divergent graphs [LNP06], etc.

23 ► `ComplementGraph(` *G* `)` A

Returns the new graph H such that $V(H) = V(G)$ and $xy \in E(H) \iff xy \notin E(G)$.

```
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 ] ] )
```

24 ► `CompleteBipartiteGraph(` *n*`,` *m* `)` F

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

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

25 ► `CompleteGraph(` *n* `)` F

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

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

26 ▶ `CompletelyParedGraph(` *G* `)` O

Returns the completely pared graph of *G*, which is obtained by repeatedly applying `ParedGraph` until no more dominated vertices remain.

```
gap> g:=PathGraph(6);
Graph( Category := SimpleGraphs, Order := 6, Size := 5, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4, 6 ], [ 5 ] ] )
gap> CompletelyParedGraph(g);
Graph( Category := SimpleGraphs, Order := 1, Size := 0, Adjacencies :=
[ [  ] ] )
```

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

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

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

28 ▶ `CompletesOfGivenOrder(` *G* `,` *Ord* `)` O

Returns the list of vertex sets of all complete subgraphs of order *Ord* of *G*.

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

29 ▶ `Composition(` *G* `,` *H* `)` O

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

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

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

30 ▶ `Cone(` *G* `)` O

Returns the cone of graph G. The cone of G is the graph obtained from G by adding a new vertex which is adjacent to every vertex of G. The new vertex is the first one in the new graph.

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

31 ▶ `ConnectedComponents(` *G* `)` A

Returns the connected components of *G*.

32 ▶ `ConnectedGraphsOfGivenOrder(` *n* `)` O

Returns the list of all connected graphs of order *n* (upto isomorphism). This operation uses Brendan McKay's data published here:

`https://cs.anu.edu.au/people/Brendan.McKay/data/graphs.html` .

These data are included with the YAGS distribution in its `data` directory. Hence this operation simply reads the corresponding file in that directory using `ImportGraph6(` *Filename* `)`. Therefore, the integer *n* must be in the range from 1 upto 9. Data for graphs on 10 vertices is also available, but not included with YAGS, it may not be practical to use that data, but if you would like to try, all you have to do is to copy (and to uncompress) the corresponding file into the directory *YAGS-Directory*`/data`.

```
gap> ConnectedGraphsOfGivenOrder(3);
[ Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
    [ [ 3 ], [ 3 ], [ 1, 2 ] ] ), Graph( Category := SimpleGraphs, Order :=
    3, Size := 3, Adjacencies := [ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ] ) ]
gap> ConnectedGraphsOfGivenOrder(4);
[ Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
    [ [ 4 ], [ 4 ], [ 4 ], [ 1, 2, 3 ] ] ),
  Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
    [ [ 3, 4 ], [ 4 ], [ 1 ], [ 1, 2 ] ] ),
  Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
    [ [ 3, 4 ], [ 4 ], [ 1, 4 ], [ 1, 2, 3 ] ] ),
  Graph( Category := SimpleGraphs, Order := 4, Size := 4, Adjacencies :=
    [ [ 3, 4 ], [ 3, 4 ], [ 1, 2 ], [ 1, 2 ] ] ),
  Graph( Category := SimpleGraphs, Order := 4, Size := 5, Adjacencies :=
    [ [ 3, 4 ], [ 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] ),
  Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
    [ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] ) ]
gap> Length(ConnectedGraphsOfGivenOrder(9));
261080
gap> ConnectedGraphsOfGivenOrder(10);
#W Unreadable File: /opt/gap4r7/pkg/yags/data/graph10c.g6
fail
```

33 ▶ `Coordinates(` *G* `)` O

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

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

34 ▶ `CopyGraph(` *G* `)` O

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

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

35 ▶ `CuadraticRingGraph(` *Rng* `)` O

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

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

36 ▶ `Cube` V

The 1-skeleton of Plato's cube.

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

37 ▶ `CubeGraph(` *n* `)` F

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

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

38 ▶ `CycleGraph(` *n* `)` F

Returns the cyclic graph on n vertices.

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

39 ▶ `CylinderGraph(` *b*`,` *h* `)` F

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

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

40 ► `DartGraph` V

A diamond with a pendant vertex and maximum degree 4.

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

41 ► `DeclareQtfyProperty(` *Name*, *Filter* `)` F

For internal use.

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

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

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

42 ▸ `Diameter(` G `)` A

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

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

43 ▸ `DiamondGraph` V

The graph on 4 vertices and 5 edges.

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

44 ▸ `DiscreteGraph(` n `)` F

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

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

45 ▸ `DisjointUnion(` G, H `)` O

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

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

46 ▸ `Distance(` G, x, y `)` O

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

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

47 ▸ `Distances(` G, A, B `)` O

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

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

48 ► `DistanceGraph(` *G*, *Dist* `)` O

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

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

49 ► `DistanceMatrix(` *G* `)` A

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

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

50 ► `DistanceSet(` *G*, *A*, *B* `)` O

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

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

51 ► `Dodecahedron` V

The 1-skeleton of Plato's Dodecahedron.

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

52 ► `DominatedVertices(` G `)` A

Returns the set of dominated vertices of G.

A vertex x is dominated by another vertex y when the closed neighborhood of x is contained in that of y. However, when there are twin vertices (mutually dominated vertices), exactly one of them (in each equivalent class of mutually dominated vertices) does not appear in the returned set.

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

53 ► `DominoGraph` V

Two squares glued by an edge.

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

54 ► `Draw(` G `)` O

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

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

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

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

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

55 ▶ `DumpObject(` *Obj* `)` O

Dumps all information available for object *Obj*. 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" ] )
```

56 ▶ `EasyExec(` *Dir*, *ProgName*, *InString* `)` O
▶ `EasyExec(` *ProgName*, *InString* `)` O

Calls external program *ProgName* located in directory *Dir*, feeding it with *InString* as input and returning the output of the external program as a string. *Dir* must be a directory object or a list of diretory objects. If *Dir* is not provided, *ProgName* must be in the system's binary PATH. 'fail' is returned if the program could not be located.

```
gap> s:=EasyExec("date","");;
gap> s;
"Sun Nov  9 10:36:16 CST 2014\n"
gap> s:=EasyExec("sort","4\n2\n3\n1");;
gap> s;
"1\n2\n3\n4\n"
```

Currently, this operation is not working on MS Windows.

57 ▶ `Eccentricity(` *G*, *x* `)` F

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

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

58 ▶ `Edges(` *G* `)` O

Returns the list of edges of graph G in the case of `SimpleGraphs`.

```
gap> g1:=CompleteGraph(3);
Graph( Category := SimpleGraphs, Order := 3, Size := 3, Adjacencies :=
[ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ] )
gap> Edges(g1);
[ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ] ]
```

In the case of `UndirectedGraphs`, it also returns the loops. While in the other categories, `Edges` actually does not return the edges, but the loops and arrows of G.

```
gap> g2:=CompleteGraph(3:GraphCategory:=UndirectedGraphs);
Graph( Category := UndirectedGraphs, Order := 3, Size := 6, Adjacencies :=
[ [ 1, 2, 3 ], [ 1, 2, 3 ], [ 1, 2, 3 ] ] )
gap> Edges(g2);
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 2, 2 ], [ 2, 3 ], [ 3, 3 ] ]
gap> g3:=CompleteGraph(3:GraphCategory:=Graphs);
Graph( Category := Graphs, Order := 3, Size := 9, Adjacencies :=
[ [ 1, 2, 3 ], [ 1, 2, 3 ], [ 1, 2, 3 ] ] )
gap> Edges(g3);
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 2, 1 ], [ 2, 2 ], [ 2, 3 ], [ 3, 1 ],
  [ 3, 2 ], [ 3, 3 ] ]
```

59 ► `FanGraph(` *n* `)` F

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

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

60 ► `FishGraph` V

A square and a triangle glued by a vertex.

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

61 ► `GemGraph` V

The 3-Fan graph.

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

62 ► `Girth(` *G* `)` A

Returns the length of the minimum induced cycle in G. At this time, this works only when G belongs to the graph categories `SimpleGraphs` or `UndirectedGraphs`. If G has loops, its girth is 1 by definition.

```
gap> Girth(Octahedron);
3
gap> Girth(PetersenGraph);
5
gap> Girth(Cube);
4
gap> Girth(PathGraph(5));
infinity
gap> g:=AddEdges(CycleGraph(4),[[3,3]]:GraphCategory:=UndirectedGraphs);
Graph( Category := UndirectedGraphs, Order := 4, Size := 5, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 3, 4 ], [ 1, 3 ] ] )
gap> Girth(g);
1
```

63 ▶ `Graph(` *Rec* `)` O

Returns a new graph created from the record *Rec*. 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 ] ] )
```

64 ▶ `GraphAttributeStatistics(` *OrderList*`,` *ProbList*`,` *Attribute* `)` F

Returns statistics for graph attribute *Attribute*. For each of the orders n in *OrderList* and for each of the probabilities p in *ProbList* this function generates 100 random graphs of order n and edge probability p and then evaluates the graph attribute *Attribute* on each of them. The function then returns statistical data on these experiments. The form in which the statistical data is reported depend on a number of issues and is best explained by examples.

First let us consider the case where *Attribute* is a Boolean attribute (always returns `true` or `false`) and where *OrderList* and *ProbList* consist of a unique value. In this case, the respective lists may be replaced by the corresponding unique values on invocation:

```
gap> GraphAttributeStatistics(10,1/2,IsCliqueHelly);
43
```

This tells us that 43 of the 100 examined random graphs resulted to be clique-Helly; The random sample was constructed using graphs of order 10 and edge probability 1/2.

Now we can specify a list of probabilities to be examined:

```
gap> GraphAttributeStatistics(10,1/10*[1..9],IsCliqueHelly);
[ 100, 100, 95, 77, 36, 22, 41, 72, 97 ]
```

The last example tells us that, for graphs on 10 vertices, the property IsCliqueHelly is least probable to be true for graphs with edge probabilities 5/10 6/10 and 7/10, being 6/10 the probability that reaches the minimum in the random sample. Note that the 36 in the previous example does not match the 43 in the first one, this is to be expected as the statistics are compiled from a random sample of graphs. Also, note that in the previous example, 900 random graphs where generated and examined.

We can also specify a list of orders to consider:

```
gap> GraphAttributeStatistics([10,12..20],1/10*[1..9],IsCliqueHelly);
[ [ 100, 100, 91, 63, 30, 23, 39, 65, 99 ], [ 100, 98, 81, 35, 4, 2, 20, 63, 98 ]
    , [ 100, 95, 49, 15, 1, 2, 13, 51, 98 ], [ 99, 82, 39, 3, 0, 2, 9, 42, 97 ],
  [ 100, 86, 15, 0, 0, 0, 7, 32, 93 ], [ 100, 69, 5, 0, 0, 0, 3, 24, 90 ] ]
gap> Display(last);
[ [  100,  100,   91,   63,   30,   23,   39,   65,   99 ],
  [  100,   98,   81,   35,    4,    2,   20,   63,   98 ],
  [  100,   95,   49,   15,    1,    2,   13,   51,   98 ],
  [   99,   82,   39,    3,    0,    2,    9,   42,   97 ],
  [  100,   86,   15,    0,    0,    0,    7,   32,   93 ],
  [  100,   69,    5,    0,    0,    0,    3,   24,   90 ] ]
```

Which tell us that the observed bimodal distribution is even more pronounced when the order of the graphs considered grows.

In the case of a non-Boolean attribute `GraphAttributeStatistics()` reports the values that *Attribute* took on the sample as well as the number of times that each of these values where obtained:

```
gap> GraphAttributeStatistics(10,1/2,Diameter);
[ [ 2, 26 ], [ 3, 60 ], [ 4, 8 ], [ 6, 1 ], [ infinity, 5 ] ]
```

The returned statistics mean that among the 100 generated random graphs on 10 vertices with edge probability 1/2, there were 26 graphs with diameter 2, 60 graphs of diameter 3, 8 of 4, 1 of 6 and 5 graphs which were not connected.

Now it should be evident the format of the returned statistics when we specify a list of probabilities and/or a list of orders to be considered for a non-Boolean Attribute:

```
gap> GraphAttributeStatistics(10,1/5*[1..4],Diameter);
[ [ [ 3, 3 ], [ 4, 5 ], [ 5, 9 ], [ 6, 3 ], [ 7, 2 ], [ infinity, 78 ] ],
  [ [ 2, 8 ], [ 3, 55 ], [ 4, 19 ], [ 5, 3 ], [ infinity, 15 ] ],
  [ [ 2, 73 ], [ 3, 26 ], [ 4, 1 ] ], [ [ 2, 100 ] ] ]
gap> GraphAttributeStatistics([10,12,14],1/5*[1..4],Diameter);
[ [ [ [ 4, 8 ], [ 5, 7 ], [ 6, 3 ], [ infinity, 82 ] ],
      [ [ 2, 3 ], [ 3, 64 ], [ 4, 15 ], [ 5, 3 ], [ infinity, 15 ] ],
      [ [ 2, 69 ], [ 3, 30 ], [ infinity, 1 ] ], [ [ 2, 100 ] ] ],
  [ [ [ 3, 1 ], [ 4, 11 ], [ 5, 13 ], [ 6, 7 ], [ 7, 3 ], [ 8, 2 ],
          [ infinity, 63 ] ],
      [ [ 2, 8 ], [ 3, 69 ], [ 4, 18 ], [ 5, 2 ], [ infinity, 3 ] ],
      [ [ 2, 79 ], [ 3, 21 ] ], [ [ 2, 100 ] ] ],
  [ [ [ 3, 1 ], [ 4, 15 ], [ 5, 13 ], [ 6, 5 ], [ 7, 4 ], [ 8, 3 ],
          [ infinity, 59 ] ], [ [ 2, 6 ], [ 3, 82 ], [ 4, 9 ], [ infinity, 3 ] ],
      [ [ 2, 86 ], [ 3, 14 ] ], [ [ 2, 100 ] ] ] ]
```

65 ► `Graph6ToGraph(` *String* `)` O

Returns the graph represented by *String* which is encoded using Brendan McKay's graph6 format. This operation allows us to read data in databases which use this format. Several such databases can be found here:

`https://cs.anu.edu.au/people/Brendan.McKay/data/graphs.html` .

The graph6 format is described here:

`https://cs.anu.edu.au/people/Brendan.McKay/data/formats.txt` .

```
gap> Graph6ToGraph("D?{");
Graph( Category := SimpleGraphs, Order := 5, Size := 4, Adjacencies :=
[ [ 5 ], [ 5 ], [ 5 ], [ 5 ], [ 1, 2, 3, 4 ] ] )
gap> Graph6ToGraph("FUzvW");
Graph( Category := SimpleGraphs, Order := 7, Size := 15, Adjacencies :=
[ [ 3, 4, 5, 6, 7 ], [ 4, 5, 6, 7 ], [ 1, 5, 6, 7 ], [ 1, 2, 6 ],
  [ 1, 2, 3, 7 ], [ 1, 2, 3, 4, 7 ], [ 1, 2, 3, 5, 6 ] ] )
gap> Graph6ToGraph("HUzv~z}");
Graph( Category := SimpleGraphs, Order := 9, Size := 29, Adjacencies :=
[ [ 3, 4, 5, 6, 7, 8, 9 ], [ 4, 5, 6, 7, 8, 9 ], [ 1, 5, 6, 7, 8, 9 ],
  [ 1, 2, 6, 7, 8, 9 ], [ 1, 2, 3, 7, 8, 9 ], [ 1, 2, 3, 4, 7, 8, 9 ],
  [ 1, 2, 3, 4, 5, 6, 9 ], [ 1, 2, 3, 4, 5, 6 ], [ 1, 2, 3, 4, 5, 6, 7 ] ] )
```

See also `ImportGraph6(` *Filename* `)`.

66 ► `GraphByAdjacencies(` *AdjList* `)` F

Returns a new graph having *AdjList* 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 ] ] )
```

67 ► `GraphByAdjMatrix(` *Mat* `)` F

Returns a new graph created from an adjacency matrix *Mat*. The matrix *Mat* 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 ] ]
```

68 ► `GraphByCompleteCover(` *Cover* `)` F

Returns the minimal graph where the elements of *Cover* 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 ] ] )
```

69 ▶ `GraphByEdges(` *L* `)` F

Returns the minimal graph such that the pairs in *L* are edges.

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

The vertices of the constructed graph range from 1 to the maximum of the numbers appearing in *L*.

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

Note that `GraphByWalks` has an even greater functionality.

70 ▶ `GraphByRelation(` *V* `,` *Rel* `)` F
▶ `GraphByRelation(` *n* `,` *Rel* `)` F

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

```
gap> Rel:=function(x,y) return Intersection(x,y)<>[]; end;;
gap> GraphByRelation([[1,2,3],[3,4,5],[5,6,7]],Rel);
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 ] ] )
```

71 ▶ `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 ] ] )
```

The vertices in the constructed graph range from 1 to the maximum of the numbers appearing in *Walk1*, *Walk2*, ... etc.

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

72 ▶ `GraphCategory( [ G, ... ] )` F

For internal use. Returns the minimal common category to a list of graphs. If the list of graphs is empty, the default category is returned.

The partial order (by inclusion) among graph categories is as follows:

$$\texttt{SimpleGraphs} < \texttt{UndirectedGraphs} < \texttt{Graphs},$$

$$\texttt{OrientedGraphs} < \texttt{LooplessGraphs} < \texttt{Graphs},$$

$$\texttt{SimpleGraphs} < \texttt{LooplessGraphs} < \texttt{Graphs}$$

```
gap> g1:=CompleteGraph(2:GraphCategory:=SimpleGraphs);
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies :=
[ [ 2 ], [ 1 ] ] )
gap> g2:=CompleteGraph(2:GraphCategory:=OrientedGraphs);
Graph( Category := OrientedGraphs, Order := 2, Size := 1, Adjacencies :=
[ [ 2 ], [  ] ] )
gap> g3:=CompleteGraph(2:GraphCategory:=UndirectedGraphs);
Graph( Category := UndirectedGraphs, Order := 2, Size := 3, Adjacencies :=
[ [ 1, 2 ], [ 1, 2 ] ] )
gap> GraphCategory([g1,g2,g3]);
<Operation "Graphs">
gap> GraphCategory([g1,g2]);
<Operation "LooplessGraphs">
gap> GraphCategory([g1,g3]);
<Operation "UndirectedGraphs">
```

73 ▶ `Graphs( )` C

`Graphs` is the most general graph category in YAGS. This category contains all graphs that can be represented in YAGS. A graph in this category may contain loops, arrows and edges (which in YAGS are exactly the same as two opposite arrows between some pair of vertices). This graph category has no parent category.

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

74 ▶ `GraphsOfGivenOrder( n )` O

Returns the list of all graphs of order n (upto isomorphism). This operation uses Brendan McKay's data published here:

`https://cs.anu.edu.au/people/Brendan.McKay/data/graphs.html` .

These data are included with the YAGS distribution in its `data` directory. Hence this operation simply reads the corresponding file in that directory using `ImportGraph6( Filename )`. Therefore, the integer n must be in the range from 1 upto 9. Data for graphs on 10 vertices is also available, but not included with YAGS, it may not be practical to use that data, but if you would like to try, all you have to do is to copy (and to uncompress) the corresponding file into the directory *YAGS-Directory*/`data`.

```
gap> GraphsOfGivenOrder(2);
[ Graph( Category := SimpleGraphs, Order := 2, Size := 0, Adjacencies :=
    [ [  ], [  ] ] ), Graph( Category := SimpleGraphs, Order := 2, Size :=
    1, Adjacencies := [ [ 2 ], [ 1 ] ] ) ]
gap> GraphsOfGivenOrder(3);
[ Graph( Category := SimpleGraphs, Order := 3, Size := 0, Adjacencies :=
    [ [  ], [  ], [  ] ] ), Graph( Category := SimpleGraphs, Order :=
    3, Size := 1, Adjacencies := [ [ 3 ], [  ], [ 1 ] ] ),
  Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
    [ [ 3 ], [ 3 ], [ 1, 2 ] ] ), Graph( Category := SimpleGraphs, Order :=
    3, Size := 3, Adjacencies := [ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ] ) ]
gap> Length(GraphsOfGivenOrder(9));
274668
gap> GraphsOfGivenOrder(10);
#W Unreadable File: /opt/gap4r7/pkg/yags/data/graph10.g6
fail
```

75 ► `GraphSum(` *G*, *L* `)` O

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

The lexicographic sum is computed as follows:

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

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

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

76 ► `GraphToRaw(` *FileName*, *G* `)` O

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

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

77 ► `GraphUpdateFromRaw(` *FileName*, *G* `)` O

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

78 ► `GroupGraph(` *G*, *Grp*, *Act* `)` O
► `GroupGraph(` *G*, *Grp* `)` O

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

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

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

79 ▶ `HararyToMcKay(` *Spec* `)` O
▶ `McKayToHarary(` *index* `)` O

Returns the McKay's *index* of a Harary's graph specification *Spec* and viceversa. Frank Harary published in his book [Har69], a list af all 208 simple graphs of order upto 6 (upto isomorphism). Each of them had a label (which we call *Harary's graph specification*) of the form `[` *n*`,` *m*`,` *s* `]` where *n* is the number of vertices, *m* is the number of edges, and *s* is a consecutive integer which uniquely identifies the graph from the others with the same *n* and *m*. On the other hand, Brendan McKay published data sets containing a list of all graphs of order upto 10 (also upto isomorphism), here:

`https://cs.anu.edu.au/people/Brendan.McKay/data/graphs.html`

Each graph in these data sets appears in some specific position (which we call *McKay's index*).

We found it convenient to have an automated way to convert from Harary's graph specifications to McKay's indexes and viceversa.

```
gap> HararyToMcKay([1,0,1]);
1
gap> HararyToMcKay([1,0,2]);
fail
gap> HararyToMcKay([5,5,2]);
31
gap> HararyToMcKay([5,5,3]);
34
gap> HararyToMcKay([5,5,5]);
30
gap> HararyToMcKay([5,5,6]);
45
gap> HararyToMcKay([5,5,7]);
fail
gap> HararyToMcKay([6,15,1]);
208
gap> HararyToMcKay([6,15,2]);
fail
gap> List([1..208],McKayToHarary);
[ [ 1, 0, 1 ], [ 2, 0, 1 ], [ 2, 1, 1 ], [ 3, 0, 1 ], [ 3, 1, 1 ],
  [ 3, 2, 1 ], [ 3, 3, 1 ], [ 4, 0, 1 ], [ 4, 1, 1 ], [ 4, 2, 1 ],
  [ 4, 3, 3 ], [ 4, 2, 2 ], [ 4, 3, 1 ], [ 4, 3, 2 ], [ 4, 4, 1 ],

                --- many more lines here ---

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

80 ▶ `HouseGraph` V

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

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

81 ▶ `Icosahedron` V

The 1-skeleton of Plato's icosahedron.

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

82 ▶ `ImportGraph6(` *Filename* `)` O

Returns the list of graphs represented in *Filename* which are encoded using Brendan McKay's graph6 format. This operation allows us to read data in databases which use this format. Several such databases can be found here:

`https://cs.anu.edu.au/people/Brendan.McKay/data/graphs.html` .

The graph6 format is described here:

`https://cs.anu.edu.au/people/Brendan.McKay/data/formats.txt` .

The following example assumes that you have a file named `graph3.g6` in your working directory which encodes graphs in graph6 format; the contents of this file is assumed to be as indicated after the first command in the example.

```
gap> Exec("cat graph3.g6");
B?
BO
BW
Bw
gap> ImportGraph6("graph3.g6");
[ Graph( Category := SimpleGraphs, Order := 3, Size := 0, Adjacencies :=
    [ [  ], [  ], [  ] ] ), Graph( Category := SimpleGraphs, Order :=
    3, Size := 1, Adjacencies := [ [ 3 ], [  ], [ 1 ] ] ),
  Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
    [ [ 3 ], [ 3 ], [ 1, 2 ] ] ), Graph( Category := SimpleGraphs, Order :=
   3, Size := 3, Adjacencies := [ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ] ) ]
```

83 ▶ `in(` *G*`,` *Catgy* `)` O

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

84 ▶ `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 ] ] )
```

85 ▶ `InNeigh(` G`,` x `)` O

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

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

86 ▶ `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 ] ] )
```

87 ▶ `IsBoolean(` *Obj* `)` F

Returns `true` if object *Obj* is `true` or `false` and `false` otherwise.

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

88 ▶ `IsCliqueGated(` G `)` P

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

89 ▶ `IsCliqueHelly(` G `)` P

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

The Helly property is defined as follows:

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

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

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

90 ▶ `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
```

91 ▶ `IsCompleteGraph(` *G* `)` P

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

92 ▶ `IsDiamondFree(` *G* `)` P

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

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

93 ▶ `IsEdge(` *G*`,` *x*`,` *y* `)` O
▶ `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),[1,2]);
true
gap> IsEdge(PathGraph(3),1,3);
false
gap> IsEdge(PathGraph(3),[1,3]);
false
```

The first form, IsEdge(*G*, *x*, *y*), is a bit faster and hence more suitable for use in algoritms which make extensive use of this operation. On the other hand, the first form does no error checking at all, and hence, it may produce an error where the second form returns false (for instance when *x* is not a vertex of *G*). The second form is therefore a bit slower, but more robust.

```
gap> IsEdge(PathGraph(3),[7,3]);
false
gap> IsEdge(PathGraph(3),7,3);
Error, List Element: <list>[7] must have an assigned value in
  return AdjMatrix( G )[x][y]; called from
<function "unknown">( <arguments> )
 called from read-eval loop at line 4 of *stdin*
you can 'return;' after assigning a value
brk>
```

94 ▶ `IsIsomorphicGraph(` *G*`,` *H* `)` O

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

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

95 ▶ `IsLoopless(` G `)` P

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

96 ▶ `IsoMorphism(` G, H `)` O

Returns one isomorphism from G to H or `fail` if none exists. If G has n vertices, an isomorphisms $f : G \to H$ is represented as the list F=`[f(1), f(2), ..., f(n)]`.

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

See `NextIsoMorphism(` G, H, F `)`.

97 ▶ `IsoMorphisms(` G, H `)` O

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

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

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

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

99 ▶ `IsSimple(` G `)` P

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

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

100 ▶ `IsTournament(` G `)` P

Returns `true` if G is a tournament.

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

101 ▶ `IsTransitiveTournament(` G `)` P

Returns `true` if G is a transitive tournament.

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

102 ► `IsUndirected(` G `)` P

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

103 ► `JohnsonGraph(` n, r `)` F

Returns the Johnson graph $J(n,r)$. The Johnson Graph is the graph whose vertices are r-subset of the set $\{1,2,\ldots,n\}$, two of them being adjacent iff they intersect in exactly r-1 elements.

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

104 ► `Join(` G, H `)` O

Returns the join graph $G+H$ of G and H (also known as the Zykov sum); it is the graph obtained from the disjoint union of G and H by adding every possible edge from every vertex in G to every vertex in H.

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

105 ► `KiteGraph` V

A diamond with a pendant vertex and maximum degree 3.

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

106 ► `LineGraph(` G `)` O

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

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

107 ► `Link(` G, x `)` O

Returns the subgraph of G induced by the neighbors of x.

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

108 ▶ `Links(` *G* `)` A

Returns the list of subgraphs of *G* induced by the neighbors of each vertex of *G*.

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

109 ▶ `LooplessGraphs( )` C

`LooplessGraphs` is a graph category in YAGS. A graph in this category may contain arrows and edges but no loops. The parent of this category is `Graphs`.

```
gap> GraphByWalks([1,1],[1,2],[2,1],[3,2]:GraphCategory:=Graphs);
Graph( Category := Graphs, Order := 3, Size := 4, Adjacencies :=
[ [ 1, 2 ], [ 1 ], [ 2 ] ] )
gap> GraphByWalks([1,1],[1,2],[2,1],[3,2]:GraphCategory:=LooplessGraphs);
Graph( Category := LooplessGraphs, Order := 3, Size := 3, Adjacencies :=
[ [ 2 ], [ 1 ], [ 2 ] ] )
```

110 ▶ `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
```

111 ▶ `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
```

112 ► `NextIsoMorphism(` *G*, *H*, *F* `)` O

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

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

113 ► `NextPropertyMorphism(` *G*, *H*, *F*, *PropList* `)` O

Returns the next morphism (in lexicographic order) from *G* to *H* satisfying the list of properties *PropList* starting with (possibly incomplete) morphism *F*. The morphism found will me returned **and** stored in *F* in order to use it as the next starting point, in case `NextPropertyMorphism` is called again. The operation returns `fail` if there are no more morphisms of the specified type.

A number of preprogrammed properties are provided by YAGS, and the user may create additional ones. The properties provided are: `CHK_WEAK`, `CHK_MORPH`, `CHK_METRIC`, `CHK_CMPLT`, `CHK_MONO` and `CHK_EPI`.

If *G* has *n* vertices and $f : G \to H$ is a morphism, it is represented as *F*=`[f(1), f(2), ..., f(n)]`.

```
gap> g:=CycleGraph(4);;h:=CompleteBipartiteGraph(2,2);;
gap> f:=[];; PropList:=[CHK_MORPH,CHK_MONO];;
gap> NextPropertyMorphism(g,h,f,PropList);
[ 1, 3, 2, 4 ]
gap> NextPropertyMorphism(g,h,f,PropList);
[ 1, 4, 2, 3 ]
gap> NextPropertyMorphism(g,h,f,PropList);
[ 2, 3, 1, 4 ]
gap> NextPropertyMorphism(g,h,f,PropList);
[ 2, 4, 1, 3 ]
gap> NextPropertyMorphism(g,h,f,PropList);
[ 3, 1, 4, 2 ]
gap> NextPropertyMorphism(g,h,f,PropList);
[ 3, 2, 4, 1 ]
gap> NextPropertyMorphism(g,h,f,PropList);
[ 4, 1, 3, 2 ]
gap> NextPropertyMorphism(g,h,f,PropList);
[ 4, 2, 3, 1 ]
gap> NextPropertyMorphism(g,h,f,PropList);
fail
```

114 ▶ `NumberOfCliques(` G `)` A
▶ `NumberOfCliques(` G, *maxNumCli* `)` O

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

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

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

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

115 ▶ `NumberOfConnectedComponents(` G `)` A

Returns the number of connected components of G.

116 ▶ `OctahedralGraph(` n `)` F

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

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

117 ▶ `Octahedron` V

The 1-skeleton of Plato's octahedron.

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

118 ▶ `Order(` G `)` A

Returns the number of vertices, of graph G.

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

119 ▶ `OrientedGraphs( )` C

`OrientedGraphs` is a graph category in YAGS. A graph in this category may contain arrows, but no loops or edges. The parent of this category is `LooplessGraphs`.

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

120 ▶ `OutNeigh(` *G*, *x* `)` O

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

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

121 ▶ `ParachuteGraph` V

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

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

122 ▶ `ParapluieGraph` V

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

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

123 ▶ `ParedGraph(` *G* `)` O

Returns the pared graph of *G*. This is the induced subgraph obtained from *G* by removing its dominated vertices. When there are twin vertices (mutually dominated vertices), exactly one of them survives the paring in each equivalent class of mutually dominated vertices.

```
gap> g1:=PathGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3 ] ] )
gap> ParedGraph(g1);
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies :=
[ [ 2 ], [ 1 ] ] )
gap> g2:=PathGraph(2);
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies :=
[ [ 2 ], [ 1 ] ] )
gap> ParedGraph(g2);
Graph( Category := SimpleGraphs, Order := 1, Size := 0, Adjacencies :=
[ [  ] ] )
```

124 ▶ `PathGraph( n )` F

Returns the path graph on n vertices.

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

125 ▶ `PawGraph` V

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

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

126 ▶ `PetersenGraph` V

The 3-regular graph on 10 vertices having girth 5.

```
gap> PetersenGraph;
Graph( Category := SimpleGraphs, Order := 10, Size := 15, Adjacencies :=
[ [ 2, 5, 6 ], [ 1, 3, 7 ], [ 2, 4, 8 ], [ 3, 5, 9 ], [ 1, 4, 10 ],
  [ 1, 8, 9 ], [ 2, 9, 10 ], [ 3, 6, 10 ], [ 4, 6, 7 ], [ 5, 7, 8 ] ] )
```

127 ▶ `PowerGraph( G, exp )` O

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

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

128 ▶ `PropertyMorphism( G, H, PropList )` O

Returns the first morphism (in lexicographic order) from G to H satisfying the list of properties *PropList*.

A number of preprogrammed properties are provided by YAGS, and the user may create additional ones. The properties provided are: `CHK_WEAK`, `CHK_MORPH`, `CHK_METRIC`, `CHK_CMPLT`, `CHK_MONO` and `CHK_EPI`.

If G has n vertices and $f : G \to H$ is a morphism, it is represented as F`=[f(1), f(2), ..., f(n)]`.

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

129 ▶ `PropertyMorphisms( G, H, PropList )` O

Returns all morphisms from G to H satisfying the list of properties *PropList*.

A number of preprogrammed properties are provided by YAGS, and the user may create additional ones. The properties provided are: `CHK_WEAK`, `CHK_MORPH`, `CHK_METRIC`, `CHK_CMPLT`, `CHK_MONO` and `CHK_EPI`.

If G has n vertices and $f : G \to H$ is a morphism, it is represented as F=`[f(1), f(2), ..., f(n)]`.

```
gap> g:=CycleGraph(4);;h:=CompleteBipartiteGraph(2,2);;
gap> PropList:=[CHK_WEAK,CHK_MONO];;
gap> PropertyMorphisms(g,h,PropList);
[ [ 1, 3, 2, 4 ], [ 1, 4, 2, 3 ], [ 2, 3, 1, 4 ], [ 2, 4, 1, 3 ],
  [ 3, 1, 4, 2 ], [ 3, 2, 4, 1 ], [ 4, 1, 3, 2 ], [ 4, 2, 3, 1 ] ]
```

130 ▶ `QtfyIsSimple(` *G* `)` A

For internal use. Returns how far is graph G from being simple.

131 ▶ `QuotientGraph(` *G*, *Part* `)` O
▶ `QuotientGraph(` *G*, *L1*, *L2* `)` O

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

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

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

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

132 ▶ `Radius(` *G* `)` A

Returns the minimal eccentricity among the vertices of graph G.

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

133 ▶ `RandomCirculant(` *n* `)` O
▶ `RandomCirculant(` *n*, *k* `)` O
▶ `RandomCirculant(` *n*, *p* `)` O

Returns a circulant on n vertices with its *jumps* selected randomly. In its third form, each possible jump has probability p of being selected. In its second form, when k is a positive integer, exactly k jumps are selected (provided there are at least k possible jumps to select from). The first form is equivalent to specifying p=1/2.

```
gap> RandomCirculant(11,2);
Graph( Category := SimpleGraphs, Order := 11, Size := 22, Adjacencies :=
[ [ 4, 6, 7, 9 ], [ 5, 7, 8, 10 ], [ 6, 8, 9, 11 ], [ 1, 7, 9, 10 ], [ 2, 8, 10, 11 ],
  [ 1, 3, 9, 11 ], [ 1, 2, 4, 10 ], [ 2, 3, 5, 11 ], [ 1, 3, 4, 6 ], [ 2, 4, 5, 7 ],
  [ 3, 5, 6, 8 ] ] )
gap> RandomCirculant(11,2);
Graph( Category := SimpleGraphs, Order := 11, Size := 22, Adjacencies :=
[ [ 2, 4, 9, 11 ], [ 1, 3, 5, 10 ], [ 2, 4, 6, 11 ], [ 1, 3, 5, 7 ], [ 2, 4, 6, 8 ],
  [ 3, 5, 7, 9 ], [ 4, 6, 8, 10 ], [ 5, 7, 9, 11 ], [ 1, 6, 8, 10 ], [ 2, 7, 9, 11 ],
  [ 1, 3, 8, 10 ] ] )
gap> RandomCirculant(11,1/2);
Graph( Category := SimpleGraphs, Order := 11, Size := 11, Adjacencies :=
[ [ 2, 11 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4, 6 ], [ 5, 7 ], [ 6, 8 ], [ 7, 9 ],
  [ 8, 10 ], [ 9, 11 ], [ 1, 10 ] ] )
gap> RandomCirculant(11,1/2);
Graph( Category := SimpleGraphs, Order := 11, Size := 44, Adjacencies :=
[ [ 2, 3, 4, 5, 8, 9, 10, 11 ], [ 1, 3, 4, 5, 6, 9, 10, 11 ],
  [ 1, 2, 4, 5, 6, 7, 10, 11 ], [ 1, 2, 3, 5, 6, 7, 8, 11 ], [ 1, 2, 3, 4, 6, 7, 8, 9 ],
  [ 2, 3, 4, 5, 7, 8, 9, 10 ], [ 3, 4, 5, 6, 8, 9, 10, 11 ], [ 1, 4, 5, 6, 7, 9, 10, 11 ],
  [ 1, 2, 5, 6, 7, 8, 10, 11 ], [ 1, 2, 3, 6, 7, 8, 9, 11 ], [ 1, 2, 3, 4, 7, 8, 9, 10 ]
 ] )
gap> RandomCirculant(11,1/2);
Graph( Category := SimpleGraphs, Order := 11, Size := 33, Adjacencies :=
[ [ 3, 4, 6, 7, 9, 10 ], [ 4, 5, 7, 8, 10, 11 ], [ 1, 5, 6, 8, 9, 11 ],
  [ 1, 2, 6, 7, 9, 10 ], [ 2, 3, 7, 8, 10, 11 ], [ 1, 3, 4, 8, 9, 11 ],
  [ 1, 2, 4, 5, 9, 10 ], [ 2, 3, 5, 6, 10, 11 ], [ 1, 3, 4, 6, 7, 11 ],
  [ 1, 2, 4, 5, 7, 8 ], [ 2, 3, 5, 6, 8, 9 ] ] )
```

134 ► `RandomGraph(` n`,` p `)` F
► `RandomGraph(` n `)` F

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

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

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

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

135 ▶ `RandomPermutation(` *n* `)` O

Returns a random permutation of the list `[1, 2, ...` *n*`]`.

```
gap> RandomPermutation(12);
(1,8,10)(2,7,9,12)(3,5,11)(4,6)
```

136 ▶ `RandomSubset(` *Set* `)` O
▶ `RandomSubset(` *Set*`,` *k* `)` O
▶ `RandomSubset(` *Set*`,` *p* `)` O

Returns a random subset of the set *Set*. When the positive integer *k* is provided, the returned subset has *k* elements (or `fail` if *Set* does not have at least *k* elements). When the probability *p* is provided, each element of *Set* has probability *p* of being selected for inclusion in the returned subset. When *k* and *p* are both missing, it is equivalent to specifying p=1/2. In the ambiguous case when the second parameter is 1, it is interpreted as the value of *k*.

```
gap> RandomSubset([1..10],5);
[ 7, 3, 10, 6, 4 ]
gap> RandomSubset([1..10],5);
[ 3, 7, 6, 9, 10 ]
gap> RandomSubset([1..10],5);
[ 3, 9, 7, 2, 6 ]
gap> RandomSubset([1..10],5);
[ 1, 2, 4, 3, 9 ]
gap> RandomSubset([1..10],1/2);
[ 1, 3, 7, 10 ]
gap> RandomSubset([1..10],1/2);
[ 1, 2, 5, 6, 7, 8, 10 ]
gap> RandomSubset([1..10],1/2);
[ 4, 5, 8, 10 ]
gap> RandomSubset([1..10],1/2);
[ 1, 4, 10 ]
```

Even if this operation is intended to be applied to sets, it does not impose this condition on its operand, and can be applied to lists as well.

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

137 ▶ `RandomlyPermuted(` *Obj* `)` O

Returns a copy of *Obj* with the order of its elements permuted randomly. Currently, the operation is implemented for lists and graphs.

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

138 ► `RemoveEdges(` *G*, *E* `)` O

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

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

139 ► `RemoveVertices(` *G*, *V* `)` O

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

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

140 ► `RGraph` V

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

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

141 ► `RingGraph(` *Rng*, *Elms* `)` O

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

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

142 ► `SetCoordinates(` *G*, *Coord* `)` O

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

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

143 ▶ `SetDefaultGraphCategory(` *Catgy* `)` F

Sets the default graph category to *Catgy*. The default graph category is used when constructing new graphs when no other graph category is indicated. New graphs are always forced to comply with the `TargetGraph-Category`, so loops may be removed, and arrows may replaced by edges or viceversa, depending on the category that the new graph belongs to.

The available graph categories are: `SimpleGraphs`, `OrientedGraphs`, `UndirectedGraphs`, `LooplessGraphs`, and `Graphs`.

```
gap> SetDefaultGraphCategory(Graphs);
gap> GraphByWalks([1,1],[1,2],[2,1],[3,2]);
Graph( Category := Graphs, Order := 3, Size := 4, Adjacencies :=
[ [ 1, 2 ], [ 1 ], [ 2 ] ] )
gap> SetDefaultGraphCategory(LooplessGraphs);
gap> GraphByWalks([1,1],[1,2],[2,1],[3,2]);
Graph( Category := LooplessGraphs, Order := 3, Size := 3, Adjacencies :=
[ [ 2 ], [ 1 ], [ 2 ] ] )
gap> SetDefaultGraphCategory(UndirectedGraphs);
gap> GraphByWalks([1,1],[1,2],[2,1],[3,2]);
Graph( Category := UndirectedGraphs, Order := 3, Size := 3, Adjacencies :=
[ [ 1, 2 ], [ 1, 3 ], [ 2 ] ] )
gap> SetDefaultGraphCategory(SimpleGraphs);
gap> GraphByWalks([1,1],[1,2],[2,1],[3,2]);
Graph( Category := SimpleGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2 ] ] )
gap> SetDefaultGraphCategory(OrientedGraphs);
gap> GraphByWalks([1,1],[1,2],[2,1],[3,2]);
Graph( Category := OrientedGraphs, Order := 3, Size := 2, Adjacencies :=
[ [ 2 ], [  ], [ 2 ] ] )
```

144 ▶ `SimpleGraphs( )` C

`SimpleGraphs` is a graph category in YAGS. A graph in this category may contain edges, but no loops or arrows. The category has two parents: `LooplessGraphs` and `UndirectedGraphs`.

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

145 ▶ `Size(` *G* `)` A

Returns the number of edges of graph *G*.

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

146 ▶ `SnubDisphenoid` V

The 1-skeleton of the 84th Johnson solid.

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

147 ▶ `SpanningForest(` *G* `)` O

Returns a spanning forest of G.

148 ▶ `SpanningForestEdges(` *G* `)` O

Returns the edges of a spanning forest of G.

149 ▶ `SpikyGraph(` *n* `)` F

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

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

150 ▶ `SunGraph(` *n* `)` F

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

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

151 ▶ `Suspension(` *G* `)` O

Returns the suspension of graph G. The suspension of G is the graph obtained from G by adding two new vertices which are adjacent to every vertex of G but not to each other. The new vertices are the first ones in the new graph.

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

152 ▶ `TargetGraphCategory( [` *G*`, ... ] )` F

For internal use. Returns the graph category indicated in the *options stack* if any, otherwise if the list of graphs provided is not empty, returns the minimal common graph category for the graphs in the list, else returns the default graph category.

The partial order (by inclusion) among graph categories is as follows:

$$\texttt{SimpleGraphs} < \texttt{UndirectedGraphs} < \texttt{Graphs},$$

`OrientedGraphs` < `LooplessGraphs` < `Graphs`

`SimpleGraphs` < `LooplessGraphs` < `Graphs`

This function is internally called by all graph constructing operations in YAGS to decide the graph category that the newly constructed graph is going to belong. New graphs are always forced to comply with the `TargetGraphCategory`, so loops may be removed, and arrows may replaced by edges or viceversa, depending on the category that the new graph belongs to.

The *options stack* is a mechanism provided by GAP to pass implicit parameters and is used by `Target-GraphCategory` so that the user may indicate the graph category she/he wants for the new graph.

```
gap> SetDefaultGraphCategory(SimpleGraphs);
gap> g1:=CompleteGraph(2);
Graph( Category := SimpleGraphs, Order := 2, Size := 1, Adjacencies :=
[ [ 2 ], [ 1 ] ] )
gap> g2:=CompleteGraph(2:GraphCategory:=OrientedGraphs);
Graph( Category := OrientedGraphs, Order := 2, Size := 1, Adjacencies :=
[ [ 2 ], [  ] ] )
gap> DisjointUnion(g1,g2);
Graph( Category := LooplessGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2 ], [ 1 ], [ 4 ], [  ] ] )
gap> DisjointUnion(g1,g2:GraphCategory:=UndirectedGraphs);
Graph( Category := UndirectedGraphs, Order := 4, Size := 2, Adjacencies :=
[ [ 2 ], [ 1 ], [ 4 ], [ 3 ] ] )
```

In the previous examples, `TargetGraphCategory` was called internally exactly once for each new graph constructed with the following parameters:

```
gap> TargetGraphCategory();
<Operation "SimpleGraphs">
gap> TargetGraphCategory(:GraphCategory:=OrientedGraphs);
<Operation "OrientedGraphs">
gap> TargetGraphCategory([g1,g2]);
<Operation "LooplessGraphs">
gap> TargetGraphCategory([g1,g2]:GraphCategory:=UndirectedGraphs);
<Operation "UndirectedGraphs">
```

153 ▶ `Tetrahedron` V

The 1-skeleton of Plato's tetrahedron.

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

154 ▶ `TimeInSeconds( )` O

Returns the time in seconds since 1970-01-01 00:00:00 UTC as an integer. This is useful to measure execution time. It can also be used to impose time constraints on the execution of algorithms. Note however that the time reported is the 'wall time', not necessarily the time spent in the process you intend to measure.

```
gap> TimeInSeconds();
1415551598
gap> K:=CliqueGraph;;
gap>  t1:=TimeInSeconds();NumberOfCliques(K(K(K(K(Icosahedron)))));TimeInSeconds()-t1;
1415551608
44644
103
```

Currently, this operation is not working on MS Windows.

155 ▶ `TimesProduct(` G, H `)` O

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

The times product is computed as follows:

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

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

156 ▶ `TorusGraph(` n, m `)` F

Returns (the underlying graph of) a triangulation of the torus on $n \cdot m$ vertices. This graphs is constructed using $\{1, 2, \ldots, n\} \times \{1, 2, \ldots, m\}$ as the vertex set; two of them being adjacent if their difference belongs to $\{(1, 0), (0, 1), (1, 1)\}$ module $\mathbb{Z}_n \times \mathbb{Z}_m$. Hence, in the category of simple graphs, TorusGraph is a 6-regular graph when $n, m \geq 3$.

```
TorusGraph(4,4);
Graph( Category := SimpleGraphs, Order := 16, Size := 48, Adjacencies :=
[ [ 2, 4, 5, 6, 13, 16 ], [ 1, 3, 6, 7, 13, 14 ], [ 2, 4, 7, 8, 14, 15 ],
  [ 1, 3, 5, 8, 15, 16 ], [ 1, 4, 6, 8, 9, 10 ], [ 1, 2, 5, 7, 10, 11 ],
  [ 2, 3, 6, 8, 11, 12 ], [ 3, 4, 5, 7, 9, 12 ], [ 5, 8, 10, 12, 13, 14 ],
  [ 5, 6, 9, 11, 14, 15 ], [ 6, 7, 10, 12, 15, 16 ], [ 7, 8, 9, 11, 13, 16 ],
  [ 1, 2, 9, 12, 14, 16 ], [ 2, 3, 9, 10, 13, 15 ], [ 3, 4, 10, 11, 14, 16 ],
  [ 1, 4, 11, 12, 13, 15 ] ] )
```

When $n, m \geq 4$, `TorusGraph(` n, m `)` is actually a Whitney triangulation: Every triangle of the graph is a face of the triagulation. The clique behavior of these graphs were extensively studied in [LN99]. However, this operation constructs the described graph for all $n, m \geq 1$.

```
gap> TorusGraph(2,4);
Graph( Category := SimpleGraphs, Order := 8, Size := 20, Adjacencies :=
[ [ 2, 4, 5, 6, 8 ], [ 1, 3, 5, 6, 7 ], [ 2, 4, 6, 7, 8 ], [ 1, 3, 5, 7, 8 ],
  [ 1, 2, 4, 6, 8 ], [ 1, 2, 3, 5, 7 ], [ 2, 3, 4, 6, 8 ], [ 1, 3, 4, 5, 7 ] ] )
gap> TorusGraph(2,3);
Graph( Category := SimpleGraphs, Order := 6, Size := 15, Adjacencies :=
[ [ 2, 3, 4, 5, 6 ], [ 1, 3, 4, 5, 6 ], [ 1, 2, 4, 5, 6 ], [ 1, 2, 3, 5, 6 ],
  [ 1, 2, 3, 4, 6 ], [ 1, 2, 3, 4, 5 ] ] )
```

Note that in these cases, `TorusGraph(` *n*, *m* `)` is not 6-regular nor a Whitney triangulation.

157 ▶ `TreeGraph(` *arity*, *depth* `)` O
▶ `TreeGraph(` *ArityList* `)` O

Returns a tree, a connected cycle-free graph. In its second form, the vertices at height k (the root vertex has height 1 here) have `ArityList[`k`]` children. In its first form, all vertices, but the leaves, have *arity* children and the height of the leaves is *depth*+1.

```
gap> TreeGraph(2,3);
Graph( Category := SimpleGraphs, Order := 15, Size := 14, Adjacencies :=
[ [ 2, 3 ], [ 1, 4, 5 ], [ 1, 6, 7 ], [ 2, 8, 9 ], [ 2, 10, 11 ], [ 3, 12, 13 ],
  [ 3, 14, 15 ], [ 4 ], [ 4 ], [ 5 ], [ 5 ], [ 6 ], [ 6 ], [ 7 ], [ 7 ] ] )
gap> TreeGraph([3,2,2]);
Graph( Category := SimpleGraphs, Order := 22, Size := 21, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 5, 6 ], [ 1, 7, 8 ], [ 1, 9, 10 ], [ 2, 11, 12 ], [ 2, 13, 14 ],
  [ 3, 15, 16 ], [ 3, 17, 18 ], [ 4, 19, 20 ], [ 4, 21, 22 ], [ 5 ], [ 5 ], [ 6 ], [ 6 ],
  [ 7 ], [ 7 ], [ 8 ], [ 8 ], [ 9 ], [ 9 ], [ 10 ], [ 10 ] ] )
```

158 ▶ `TrivialGraph` V

The one vertex graph.

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

159 ▶ `UFFind(` *UFS*, *x* `)` F

For internal use. Implements the *find* operation on the *union-find structure*.

160 ▶ `UFUnite(` *UFS*, *x*, *y* `)` F

For internal use. Implements the *unite* operation on the *union-find structure*.

161 ▶ `UndirectedGraphs( )` C

`UndirectedGraphs` is a graph category in YAGS. A graph in this category may contain edges and loops, but no arrows. The parent of this category is `Graphs`.

```
gap> GraphByWalks([1,1],[1,2],[2,1],[3,2]:GraphCategory:=Graphs);
Graph( Category := Graphs, Order := 3, Size := 4, Adjacencies :=
[ [ 1, 2 ], [ 1 ], [ 2 ] ] )
gap> GraphByWalks([1,1],[1,2],[2,1],[3,2]:GraphCategory:=UndirectedGraphs);
Graph( Category := UndirectedGraphs, Order := 3, Size := 3, Adjacencies :=
[ [ 1, 2 ], [ 1, 3 ], [ 2 ] ] )
```

162 ▶ `UnitsRingGraph(` *Rng* `)` O

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

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

163 ▶ `VertexDegree(` *G*`,` *x* `)` O

Returns the degree of vertex *x* 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
```

164 ▶ `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 ]
```

165 ▶ `VertexNames(` *G* `)` A

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

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

166 ▶ `Vertices(` *G* `)` O

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

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

167 ▶ `WheelGraph(` *n* `)` O
▶ `WheelGraph(` *n*, *r* `)` O

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

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

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

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

168 ▶ `YAGSExec(` *ProgName*, *InString* `)` O

For internal use. Calls external program *ProgName* located in directory '*YAGSDir*/bin/' feeding it with *InString* as input and returning the output of the external program as a string. 'fail' is returned if the program could not be located.

```
gap> YAGSExec("time","");
"1415551127\n"
gap> YAGSExec("nauty","l=0$=1dacn=5 g1,2,3. xbzq");
"(4,5)\n(2,3)\n[2,3,4,5,1]\n[\"cb0c\",\"484f264\",\"b0e19f1\"]\n"
```

Currently, this operation is not working on MS Windows.

169 ▶ `YAGSInfo` V

A global record where much YAGS-related information is stored. This is intended for internal use, and much of this information is undocumented, but some of the data stored here could possibly be useful for advanced users.

However, storing user information in this record and/or changing the values of the stored information is discouraged and may produce unpredictable results and an unstable system.

```
gap> YAGSInfo;
rec( AuxInfo := "/dev/null", DataDirectory := "/opt/gap4r7/pkg/yags/data",
  Directory := "/opt/gap4r7/pkg/yags", Internal := rec(  ), Version := "0.0.1",
  graph6 := rec( BinListToNum := function( L ) ... end,
      BinListToNumList := function( L ) ... end, McKayN := function( n ) ... end,
      McKayR := function( L ) ... end, NumListToString := function( L ) ... end,
      NumToBinList := function( n ) ... end, PadLeftnSplitList6 := function( L ) ... end,
      PadRightnSplitList6 := function( L ) ... end,
      StringToBinList := function( Str ) ... end ) )
```

Bibliography

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

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

[Esc73] F. Escalante. Über iterierte Clique-Graphen. *Abh. Math. Sem. Univ. Hamburg*, 39:59–68, 1973.

[FLNP13] M.E. Frías-Armenta, F. Larrión, V. Neumann-Lara, and M.A. Pizaña. Edge contraction and edge removal on iterated clique graphs. *Discrete Applied Mathematics*, 161(1011):1427 – 1439, 2013.

[FNP04] Martín E. Frías-Armenta, Víctor Neumann-Lara, and M. A. Pizaña. Dismantlings and iterated clique graphs. *Discrete Math.*, 282(1-3):263–265, 2004.

[Har69] Frank Harary. *Graph theory*. Addison-Wesley Publishing Co., Reading, Mass.-Menlo Park, Calif.-London, 1969.

[LN97] F. Larrión and V. Neumann-Lara. A family of clique divergent graphs with linear growth. *Graphs Combin.*, 13(3):263–266, 1997.

[LN99] F. Larrión and V. Neumann-Lara. Clique divergent graphs with unbounded sequence of diameters. *Discrete Math.*, 197/198:491–501, 1999.

[LN02] F. Larrión and V. Neumann-Lara. On clique-divergent graphs with linear growth. *Discrete Math.*, 245:139–153, 2002.

[LNP04] F. Larrión, V. Neumann-Lara, and M. A. Pizaña. Clique divergent clockwork graphs and partial orders. *Discrete Appl. Math.*, 141(1-3):195–207, 2004.

[LNP06] F. Larrión, V. Neumann-Lara, and M. A. Pizaña. Graph relations, clique divergence and surface triangulations. *J. Graph Theory*, 51(2):110–122, 2006.

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

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

Index

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