YAGS
Yet Another Graph System
GAP4 Package

Version 0.8
by

R. Mac Kinney Romero
M. A. Pizaña López
Electrical Engineering Department
Universidad Autonoma Metropolitana
email: rene,map@xanum.uam.mx

January 2006

Contents

1 Basics

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

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

The tutorial is divided as follows:

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

1.1 Using YAGS

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

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

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

true
```

a double semicolon can be used to avoid the banner.

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

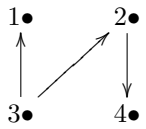
```
gap> ?RandomGraph
Help: Showing 'yags: RandomGraph'

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

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

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

1.2 Definition of graphs

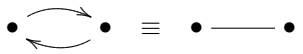
A graph is defined as follows. A graph G is a set of vertices V and a set of edges (arrows) E, $G = \{V,E\}$. The set of edges is a set of tuples of vertices (v_i, v_j) that belong to $V, v_i, v_j \in V$ representing that v_i, v_j are adjacent.

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



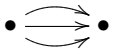
The adjacencies can also be represented as a matrix. This would be a boolean matrix M where two vertices i, j are adjacent if $M[i, j] = true$ and not adjacent otherwise.

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



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

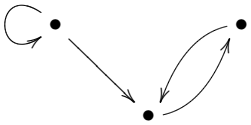
YAGS handles graphs that have arrows, edges and loops. Graphs that, for instance, have multiple arrows between vertices are not handled by YAGS .



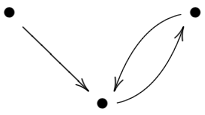
1.3 A taxonomy of graphs

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

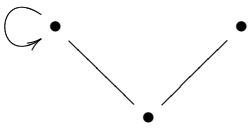
- *Graphs*: graphs with no particular property.



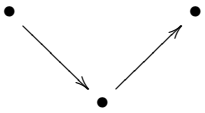
- *Loopless*: graphs with no loops.



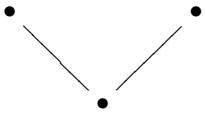
- *Undirected*: graphs with no arrows but only edges.



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



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



The following figure shows the relationships among categories.

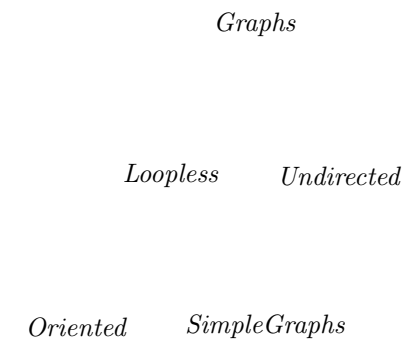


Figure 1: Graph Categories

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

Further information regarding categories can be found on section 2.

1.4 Creating Graphs

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

- Adjacency List
- Adjacency Matrix

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

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

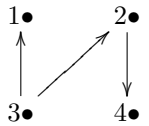
and the adjacency matrix

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

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

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

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



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

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

```
gap> g:=AdjacencyGraph([[],[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 ], [  ] ] )
```

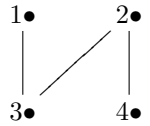
Creates the same graph. Note that we explicitly give the graph category as before. We also can use the command *AdjMatrix* as in

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

Which creates a graph with only edges

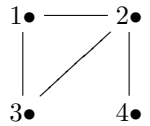


There are many functions to create graphs, some from existing graphs and some create interesting well known graphs.

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

```
gap> g:=AdjacencyGraph([[],[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 ] ] )
```

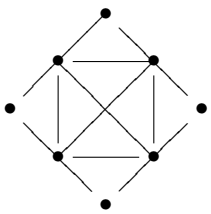
that yields the graph h



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 section 3.

1.5 Transforming graphs

1.6 Experimenting on graphs

Coming soon!

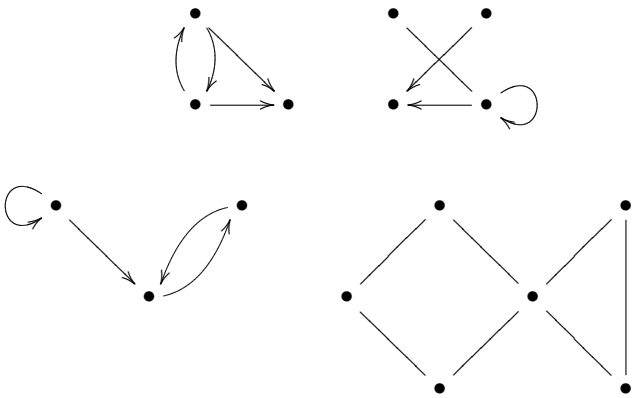
2 Categories

2.1 Graph Categories

1 ▶ `Graphs( )` C

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

Among them we can find:

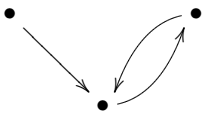


2 ▶ `LooplessGraphs( )` C

Loopless Graphs are graphs which have no loops.

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

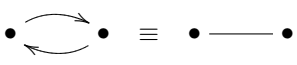
Loopless graphs have no such arrows.



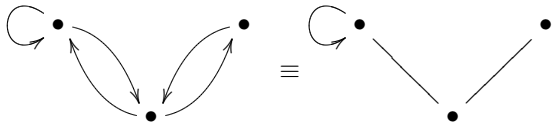
3 ▶ `UndirectedGraphs( )` C

Undirected Graphs are graphs which have no directed arrows.

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



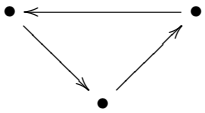
Undirected graphs have no arrows but only edges.



4 ▶ `OrientedGraphs( )` C

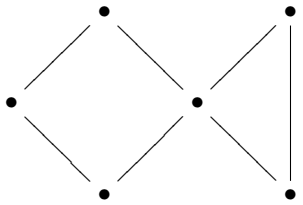
Oriented Graphs are graphs which have arrows in only one direction between any two vertices.

Oriented graphs have no edges but only arrows.



5 ▶ `SimpleGraphs( )` C

Simple Graphs are graphs with no loops and undirected.



The following figure shows the relationships among categories.

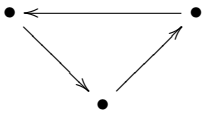
Graphs

Loopless *Undirected*

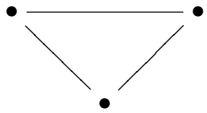
Oriented *SimpleGraphs*

Figure 1: Graph Categories

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



as a simple graph YAGS will normalize the graph as



For further examples see the following section.

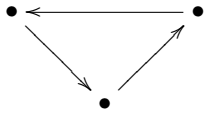
2.2 Default Category

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

1 ▶ `SetDefaultGraphCategory( C )` F

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

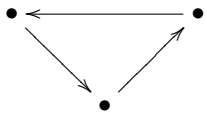
```
SetDefaultGraphCategory(Graphs);
G:=RandomGraph(4);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 3, 4 ], [ 4 ], [ 1, 2, 3, 4 ], [ 2 ] ] )
```



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

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

Now G is a simple graph.



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

2 ▶ `GraphCategory( [G, ... ] )` F

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

If the list is empty the default category is returned.

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

Returns the category which will be used to process a list of graphs. If an option category has been given it will return that category. Otherwise it will behave as Function *GraphCategory* (2.2.2). See Section 2 for the relationship among categories.

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

4 ▶ `in( G, C )` O

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

3 Constructing graphs

3.1 Primitives

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

1 ► `Graph( R )` O

Creates a graph from the record R. There are two representations from which a graph can be created:

1. From an adjacency list.
2. From an adjacencies matrix.

The record must, therefore, provide a field *Adjacencies* containing the list of adjacencies for each vertex or alternatively a field *AdjMatrix* with the adjacency matrix. Additionaly we must provide the category, or categories, to which the new graph belongs.

2 ► `AdjMatrixGraph( M )` F

Creates a graph from an adjacency matrix M. The matrix M must be a square boolean matrix. The category to which the graph created belongs is the default category. For more information on categories see Section Categories(2).

3 ► `AdjacencyGraph( A )` F

Creates a graph from a list of adjacencies A. The category to which the graph created belongs is the default category. For more information on Section Categories(2).

4 ► `CompleteCoverGraph( C )` F

Creates a complete cover graph os size C.

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

Creates a graph from a relation.

6 ► `IntersectionGraph( L )` F

Creates a graph from list L where L is an intersection list.

The following functions create graphs from existing graphs

7 ► `CopyGraph( G )` O

Creates a fresh copy of graph G.

8 ► `InducedSubgraph( G, V )` O

Creates an induced graph from graph G using only vertices V.

9 ► `RemoveVertices( G, V )` O

Creates a graph from graph G removing vertices V.

10 ▶ `AddEdges( G, E )` O

Creates a graph from graph G adding the set of edges E.

11 ▶ `RemoveEdges( G, E )` O

Creates a graph from graph G removing edges E.

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

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

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

3.2 Families

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

1 ▶ `DiscreteGraph( n )` F

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

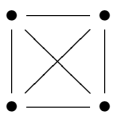
```
gap> DiscreteGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 0, Adjacencies :=
[ [  ], [  ], [  ], [  ] ] )
```

4-Discrete Graph

2 ▶ `CompleteGraph( n )` F

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

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

4-Complete Graph 

3 ▶ `PathGraph( n )` F

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

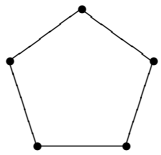
```
gap> PathGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3 ] ] )
```

4-Path Graph

4 ▶ `CycleGraph( n )` F

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

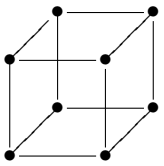
```
gap> CycleGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size := 5, Adjacencies :=
[ [ 2, 5 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 1, 4 ] ] )
```

5-Cycle Graph 

5 ▶ `CubeGraph( n )` F

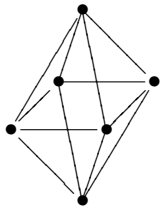
Returns a Cube Graph of order n. A cube graph is a graph where each vertex has degree n.

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

3-Cube Graph 

6 ▶ `OctahedralGraph( n )` F

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

3-Octahedral Graph 

7 ▶ `JohnsonGraph( n, r )` F

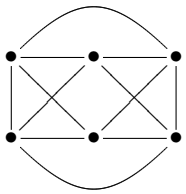
Returns a Johnson Graph $J(n, r)$· A Johnson Graph is a graph constructed as follows. Each vertex represents a subset of the set $\{1, \ldots, n\}$ with cardinality r·

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

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

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

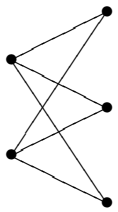
```
gap> JohnsonGraph(4,2);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 4, 6 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ],
 [ 1, 3, 4, 6 ], [ 2, 3, 4, 5 ] ] )
```

4,2-Johnson Graph 

8 ▶ `CompleteBipartiteGraph( n, m )` F

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

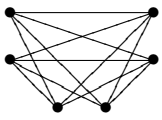
```
gap> CompleteBipartiteGraph(2,3);
Graph( Category := SimpleGraphs, Order := 5, Size := 6, Adjacencies :=
[ [ 3, 4, 5 ], [ 3, 4, 5 ], [ 1, 2 ], [ 1, 2 ], [ 1, 2 ] ] )
```

2,3-Complete Bipartite Graph 

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

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

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

2,2,2-Complete Multipartite Graph 

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

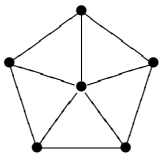
Returns a Random Graph of order n. The first form additionally takes a parameter p, the probability of an edge to exist. A probability 1 will return a Complete Graph and a probability 0 a Discrete Graph.

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

5-Random Graph

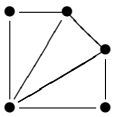
11 ▶ `WheelGraph( N )` F

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

Wheel Graph of Order 5 

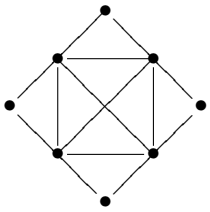
12 ▶ `FanGraph( N )` F

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

4-Fan Graph 

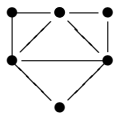
13 ▶ `SunGraph( N )` F

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

4-Sun Graph 

14 ▶ `SpikyGraph( N )` F

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

3-Spiky Graph 

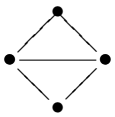
15 ▶ `TrivialGraph` V

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

Trivial Graph •

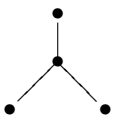
16 ▶ `DiamondGraph` V

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

Diamond Graph 

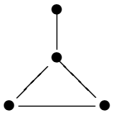
17 ▶ `ClawGraph` V

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

Claw Graph 

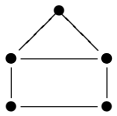
18 ▶ `PawGraph` V

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

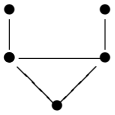
Paw Graph 

19 ▶ `HouseGraph` V

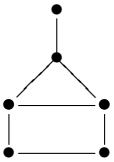
```
gap> HouseGraph;
Graph( Category := SimpleGraphs, Order := 5, Size := 6, Adjacencies :=
[ [ 2, 3 ], [ 1, 3, 4 ], [ 1, 2, 5 ], [ 2, 5 ], [ 3, 4 ] ] )
```

House Graph 

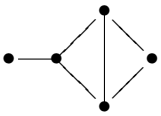
20 ▶ `BullGraph` V

Bull Graph 

21 ▶ `AntennaGraph` V

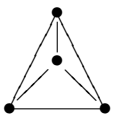
Antenna Graph 

22 ▶ `KiteGraph` V

Kite Graph 

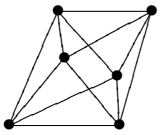
23 ▶ `Tetrahedron` V

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

Tetrahedron 

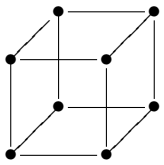
24 ▶ `Octahedron` V

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

Octahedron 

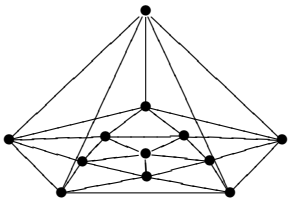
25 ▶ `Cube` V

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

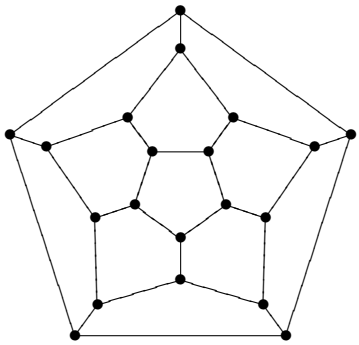
Cube Graph 

26 ▶ `Icosahedron` V

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

Icosahedron 

27 ▶ `Dodecahedron` V

Dodecahedron 

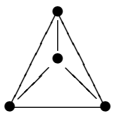
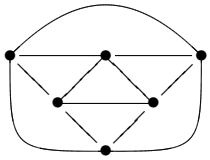
3.3 Unary operations

These are operations that can be performed over graphs.

1 ▶ `LineGraph( <G> )` O

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

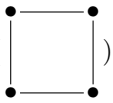
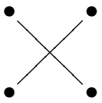
```
gap> G:=Tetrahedron;
Graph( Category := SimpleGraphs, Order := 4, Size := 6, Adjacencies :=
[ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] )
gap> LineGraph(G);
Graph( Category := SimpleGraphs, Order := 6, Size := 12, Adjacencies :=
[ [ 2, 3, 4, 5 ], [ 1, 3, 4, 6 ], [ 1, 2, 5, 6 ], [ 1, 2,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

LineGraph() = 

2 ▶ `ComplementGraph( <G> )` O

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

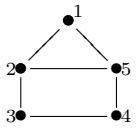
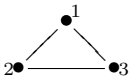
```
gap> G:=ClawGraph;
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [ 2, 3, 4 ], [ 1 ], [ 1 ], [ 1 ] ] )
gap> ComplementGraph(G);
Graph( Category := SimpleGraphs, Order := 4, Size := 3, Adjacencies :=
[ [  ], [ 3, 4 ], [ 2, 4 ], [ 2, 3 ] ] )
```

ComplementGraph() = 

3 ▶ `QuotientGraph( <G>, <P> )` O

Returns the quotient graph of graph ¡G¿ given a partition of edges ¡P¿. The quotient graph is the intersection graph of the subsets of vertives given by partition ¡P¿, ¡i.e.¿ the vertices of the quotient graph are sets of vertices given by partition ¡P¿ two of them being adjacent iff any two of the vertices in the sets are adjacent in ¡G¿.

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

QuotientGraph( , $[2,3],[4,5]$) = 

3.4 Binary operations

These are binary operations that can be performed over graphs.

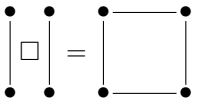
1 ▶ `BoxProduct( G, H )` O

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

The box product is calculated as follows:

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

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



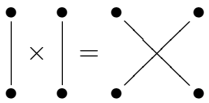
2 ▶ `TimesProduct( G, H )` O

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

The times product is computed as follows:

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

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



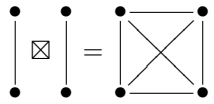
3 ▶ `BoxTimesProduct( G, H )` O

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

The box times product is calculated as follows:

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

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



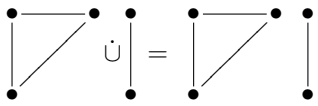
In the previous examples k^2 (*i.e.* the complete graph or order two) was chosen because it better pictures how the operators work.

4 ▶ `DisjointUnion( G, H )` O

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

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

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



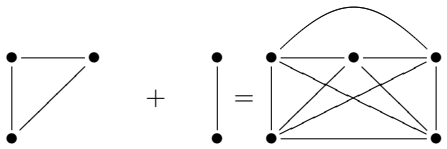
5 ▶ `Join( G, H )` O

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

Joining graphs is computed as follows:

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

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



6 ▶ `GraphSum( G, L )` O

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

The GraphSum is computed as follows:

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

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

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

7 ▶ `Composition( G, H )` O

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

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

$$G[H] = \text{GraphSum}(G, [H, \ldots, H])\cdot$$

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

4 Inspecting Graphs

4.1 Atributes and properties of graphs

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

1 ▶ `AdjMatrix( G )` A

The adjacency matrix of graph G.

2 ▶ `Order( G )` A

The order, *i.e.* number of vertices, of graph G.

3 ▶ `Size( G )` A

The size, *i.e.* number of arrows or edges of graph G.

4 ▶ `VertexNames( G )` A

A list of all vertices in graph G.

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

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

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

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

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

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

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

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

9 ▶ `CliqueNumber( G )` A

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

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

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

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

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

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

11 ▶ `IsCliqueHelly( G )` A
▶ `QtfyIsCliqueHelly( G )` P

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

The Helly property is defined as follows:

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

$$\forall \mathcal{X} \subseteq \mathcal{F}$$
$$\forall x_1, x_2 \in \mathcal{X}$$
$$x_1 \cap x_2 \neq \phi \Rightarrow \bigcap \mathcal{X} \neq \phi$$

The algorithm used to compute the Helly property is by Jayme

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

4.2 Information about graphs

The following functions give information regarding graphs.

1 ▶ `IsSimple( G )` O

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

2 ▶ `QtfyIsSimple( G )` O

Returns how far is graph G from being simple.

3 ▶ `Adjacency( G, V )` O

Returns the adjacency list of vertex V in G.

4 ▶ `Adjacencies( G )` O

Returns the adjacencies list of graph G

5 ▶ `VertexDegree( G, V )` O

Returns the degree (number of adjacencies) of vertex V in Graph G.

6 ▶ `VertexDegrees( G )` O

Returns a list with the degrees of all vertices in graph G.

7 ▶ `Edges( G )` O

Returns a list of all edges in graph G. Each edge is represented as $[i, j]$ indicating there is an edge from vertex i to vertex j.

8 ▶ `CompletesOfGivenOrder( G, o )` O

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

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

4.3 Distances

These are functions that measure distances between graphs.

1 ▶ `Distance( G, x, y )` O

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

$$d_G(x, y)$$

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

2 ▶ `DistanceMatrix( G )` A

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

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

3 ▶ `Diameter( G )` A

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

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

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

4 ▶ `Excentricity( G, x )` F

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

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

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

5 ▶ `Radius( G )` A

Returns the minimal excentricity among the vertices of graph G.

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

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

6 ▶ `Distances( G, A, B )` O

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

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

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

7 ▶ `DistanceSet( G, A, B )` O

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

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

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

8 ▶ `DistanceGraph( G, D )` O

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

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

9 ▶ `PowerGraph( G, e )` O

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

$$G^n = DistanceGraph(G, [0, 1, \ldots, e])$$

```
gap> G:=SunGraph(4);
Graph( Category := SimpleGraphs, Order := 8, Size := 14, Adjacencies :=
[ [ 2, 8 ], [ 1, 3, 4, 6, 8 ], [ 2, 4 ], [ 2, 3, 5, 6, 8 ], [ 4, 6 ],
  [ 2, 4, 5, 7, 8 ], [ 6, 8 ], [ 1, 2, 4, 6, 7 ] ] )
gap> PowerGraph(G,3);
Graph( Category := SimpleGraphs, Order := 8, Size := 28, Adjacencies :=
[ [ 2, 3, 4, 5, 6, 7, 8 ], [ 1, 3, 4, 5, 6, 7, 8 ], [ 1, 2, 4, 5, 6, 7, 8 ],
  [ 1, 2, 3, 5, 6, 7, 8 ], [ 1, 2, 3, 4, 6, 7, 8 ], [ 1, 2, 3, 4, 5, 7, 8 ],
  [ 1, 2, 3, 4, 5, 6, 8 ], [ 1, 2, 3, 4, 5, 6, 7 ] ] )
```

5 Morphisms of Graphs

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

5.1 Core Operations

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

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

Additionally it must have at least one of the following.

- CHQ_WEAK *Weak*
- CHQ_MORPH *Morph*

These properties are detailed in the next section.

1 ▶ `PropertyMorphism( G1, G2, c )` O

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

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

2 ▶ `PropertyMorphisms( G1, G2, c )` O

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

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

3 ▶ `NextPropertyMorphism( G1, G2, m, c )` O

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

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

5.2 Morphisms

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

Metric A morphism is metric if the distance (see section 4.3.1) 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 $G\cdot$

Bibliography

Index

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