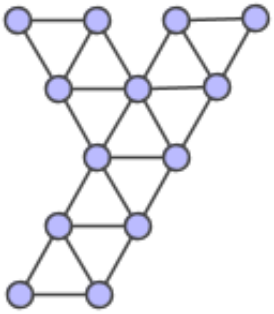
YAGS
Yet Another Graph System
Version 0.0.1

15 August 2016

R. MacKinney-Romero
M.A. Pizaña
R. Villarroel-Flores



R. MacKinney-Romero Email: rene@xanum.uam.mx

M.A. Pizaña Email: mpizana@gmail.com
Homepage: http://xamanek.izt.uam.mx/map/

R. Villarroel-Flores Email: rafaelv@uaeh.edu.mx
Homepage: http://rvf0068.github.io

Copyright

CONTACT INFORMATION:
M.A. Pizaña
yags@xamanek.izt.uam.mx
mpizana@gmail.com
Departamento de Ingeniería Eléctrica
Universidad Autónoma Metropolitana
Av. San Rafael Atlixco 186.
Col. Vicentina, Del. Iztapalapa
Ciudad de México 09340 MEXICO.

Acknowledgements

Partially supported by SEP-CONACyT, grant 183210.

We are also grateful for the support of our Universities:
Universidad Autónoma Metropolitana and Universidad Autónoma del Estado de Hidalgo.

Contents

Chapter 1

Preface

1.1 Welcome to YAGS

YAGS - *Yet Another Graph System* is a GAP package for dealing with graphs, in the sense of Graph Theory (not bar graphs, pie charts nor graphs of functions). Our graphs are then, ordered pairs $G = (V, E)$, where V is a finite set of vertices and E is a finite set of edges which are (ordered or unordered) pairs of vertices.

YAGS was initiated by M.A. Pizaña in May 2003, and soon incorporated the work of R. MacKinney-Romero and R. Villarroel-Flores. It sprang from our need of computing graphs and graph parameters within our research on graph theory and clique graphs. Consequently, YAGS is well suited for these purposes.

YAGS is a GAP package and hence its code is interpreted and not compiled (although some compilation possibilities exist in GAP). Therefore, from the very beginning, it was clear that speed is not our main goal. Instead, we wanted a very functional, full-featured system; a system adequate for rapid prototyping of algorithms; and a quick, easy-to-use, way for testing the rapidly changing working conjectures that are typical of the research process.

Over the years, YAGS grew to its present size of more than 200 methods and more than 10 thousands lines of code. We considered that all this code and effort could (and should) be useful for other people and then we decided to engage in the task of tying up loose ends and writing this manual.

We would like to mention that we started using GRAPE, and we are grateful to its author, Leonard H. Soicher, for the very useful system that we used for several years. But at some point we needed some Object-Oriented features that were not easy to implement in GRAPE and our own subsystem had to follow its own way. If the reader has a profound need for having groups acting on her/his graphs, then GRAPE may be the best choice. On the other hand, YAGS offers a much wider set of functions (Appendix B); a graph-drawing subsystem (`Draw` (B.4.15)); many methods for dealing with graph homomorphism (Chapter 5); an Object-Oriented approach that simplifies the task of working with several different graph categories (Chapter 4); and a generic backtracking subsystem useful to solve many combinatorial problems easily (Chapter 6).

1.2 Citing YAGS

If you publish a result and you used YAGS during your research, please cite us as you would normally do with a research paper:

R. MacKinney-Romero, M.A. Pizaña and R. Villarroel-Flores.
YAGS - Yet Another Graph System, Version 0.0.1 (2016)
http://xamanek.izt.uam.mx/yags/

```
@manual{YAGS,
  author = {R. MacKinney-Romero and M.A. Piza{\~n}a and R. Villarroel-Flores},
  title = {YAGS - Yet Another Graph System, Version 0.0.1},
  year = {2016},
  note = {http://xamanek.izt.uam.mx/yags/},
}
```

Several other citation formats can be obtained from the file `YAGS-DIR/CITATION` or by typing `Cite("yags");` at the GAP prompt.

1.3 Authors

The authors of YAGS in the chronological order of their first contribution are as follows:

M.A. Pizaña
Departamento de Ingeniería Eléctrica
Universidad Autónoma Metropolitana
mpizana@gmail.com

R. MacKinney-Romero
Departamento de Ingeniería Eléctrica
Universidad Autónoma Metropolitana
rene@xanum.uam.mx

R. Villarroel-Flores
Centro de Investigación en Matemáticas
Universidad Autónoma del Estado de Hidalgo
rafaelv@uaeh.edu.mx

1.4 More Information

More information about YAGS can be found on its official web page and manual:
http://xamanek.izt.uam.mx/yags/
http://xamanek.izt.uam.mx/yags/doc/chap0.html
http://xamanek.izt.uam.mx/yags/manual.pdf

You can receive notifications about YAGS (i.e. new releases, bug fixes, etc.) by subscribing to its email distribution list: http://xamanek.izt.uam.mx/yagsnews/

If you are a developer, you may contribute to our project on our public repository: https://github.com/yags/yags/

Comments, support requests, bug reports and installation notifications are welcome at yags@xamanek.izt.uam.mx.

Chapter 2

Getting Started

2.1 What is YAGS?

YAGS - *Yet Another Graph System* is a GAP package for dealing with graphs, in the sense of Graph Theory (not bar graphs, pie charts nor graphs of functions). Hence our graphs are ordered pairs $G = (V,E)$, where V is a finite set of vertices and E is a finite set of edges which are (ordered or unordered) pairs of vertices.

YAGS was designed to be useful for research on graphs theory and clique graphs. It is a very functional, full-featured system; a system adequate for rapid prototyping of algorithms; and it is a quick, easy-to-use way, for testing the rapidly changing working conjectures which are typical of the research process.

YAGS offers an ample set of functions (Appendix B); a graph-drawing subsystem (`Draw` (B.4.15)); many methods for dealing with graph homomorphism (Chapter 5); an Object-Oriented approach that simplifies the task of working with several different graph categories (Chapter 4); and a generic backtracking subsystem useful to solve many combinatorial problems easily (Chapter 6).

2.2 Installing YAGS

If you are fond of `git` and you already installed GAP, then you could clone our repository as usual (here we assume that `GAP-DIR` is your GAP installation directory):

Example

```
git clone http://github.com/yags/yags.git GAP-DIR/pkg/yags
```

Otherwise, you may follow these installation instructions:

1. Install GAP following the instructions at `http://www.gap-system.org/`.
2. Obtain YAGS from its repository `https://github.com/yags/yags/archive/master.zip`.
3. Unpack YAGS: the contents of the zip file should go under `GAP-DIR/pkg/yags/`. Here, we assume that `GAP-DIR` is your GAP installation directory.
4. Test YAGS by running GAP, loading YAGS and executing a few basic commands in a terminal:

Example

```
> gap
   --- some GAP info here ---
gap> RequirePackage("yags");

Loading  YAGS - Yet Another Graph System 0.0.1.
Copyright (C) 2016 R. MacKinney-Romero, M.A. Pizana and R. Villarroel-Flores
This is free software under GPLv3; for details type: ?yags:Copyright

true
gap> CliqueNumber(Icosahedron);NumberOfCliques(Icosahedron);
3
20
gap>
```

5. (Optional) Make us happier by sending us a brief installation notification to yags@xamanek.izt.uam.mx and subscribing to YAGS's distribution list: http://xamanek.izt.uam.mx/yagsnews/

Did it work? Congratulations! Otherwise, consider the following troubleshooting issues:

- IS GAP WORKING?
  Make sure it is. Follow carefully GAP's installation and troubleshooting procedures.

- IS THE INSTALLATION DIRECTORY CORRECT?
  The *GAP's installation directory*, GAP-DIR, is typically something like /opt/gap4r8/ (in MS Windows it may look like C:\gap4r8\). If this is the case, the *YAGS's installation directory*, YAGS-DIR, is /opt/gap4r8/pkg/yags/ (in MS Windows, it would be C:\gap4r8\pkg\yags\). Then, the full path for YAGS's info file PackageInfo.g should be /opt/gap4r8/pkg/yags/PackageInfo.g (or C:\gap4r8\pkg\yags\PackageInfo.g)

- ARE YOU USING GRAPE?
  GRAPE and YAGS are incompatible: they can not be loaded at the same time. If you had an initialization file that loads GRAPE automatically, you should disable it in order to use YAGS. Alternatively, the command gap -r starts gap disabling any user-specific configuration files.

- UNAUTHORIZED TO ACCESS GAP'S DIRECTORIES?
  The installation procedure above assumed that you have full access to your computer (i.e. that you are the root of the system or that you are using your PC or Mac). If this is not the case, you can also install YAGS under your user directory. For instance, if your user directory is /home/joe/ then you can create a subdirectory /home/joe/gaplocal/ and hence your YAGS's installation directory will be /home/joe/gaplocal/pkg/yags/. Then you can start GAP using gap -l ";/home/joe/gaplocal" so that GAP knows where your YAGS is.

2.3 A Gentle Tutorial

This tutorial assumes that you already installed GAP and YAGS; and that you have some basic understanding of GAP: user interface, the read-eval-print loop, arithmetic operations, and lists. It is strongly recommended that you have some *working directory*, WORKING-DIR, different from your

GAP's and YAGS's installation directories. For instance, if your home directory is /home/joe/ your working directory could be /home/joe/Yags/. Then you should open a terminal, move to your working directory, start GAP and then, load YAGS:

Example

```
/home/joe> cd Yags
/home/joe/Yags> gap
   --- some GAP info here ---
gap> RequirePackage("yags");

Loading  YAGS - Yet Another Graph System 0.0.1.
Copyright (C) 2016 R. MacKinney-Romero, M.A. Pizana and R. Villarroel-Flores
This is free software under GPLv3; for details type: ?yags:Copyright

true
gap>
```

The exact appearance of your system prompt (/home/joe> and /home/joe/Yags/> in the example) may be different depending on your system, but the commands 'cd Yags' and 'gap' are actually the same in all supported systems (assuming your working directory exists and is named 'Yags'). From there (starting with the command 'RequirePackage("yags");') everything happens within GAP and hence it is system-independent.

Now we want to define some graph. Say we have the list of edges of the desired graph:

$$\{\{1,2\},\{2,3\},\{3,4\},\{4,1\},\{1,5\},\{5,4\}\}$$

We can put those edges in a list and then construct the graph:

Example

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

Note that GAP uses brackets ('[' and ']') instead of braces ('{' and '}') to represent sets and lists (actually, in GAP a set is simply an ordered list). Note also that in GAP 'list' and 'List' are two different things and you can not use the latter since it is a reserved word of GAP. In general, it is better for you to use lowercase names for your variables, to avoid name clashes, since all functions in GAP and YAGS start with an uppercase letter.

The result in the previous example says that it is a graph, and a *simple graph*. By default all graphs in YAGS are simple (no loops, no arrows, no parallel edges, only plain undirected edges), in Chapter 4 we explain how to work with other types of graphs, like digraphs, loopless graphs, and graphs that may have loops (but no parallel edges are supported in YAGS at all). In this gentle tutorial all our graphs are simple.

The result also says, that the just constructed graph g have 5 vertices and 6 edges. The reported list of adjacencies means that the vertex 1 is adjacent (connected by an edge) to 2, 4 and 5, that the vertex 2 is adjacent to 1 and 3 and so on. To be sure, we can draw our graph and check if it is the intended graph:

Example

```
gap> Draw(g);
```

A separate window appears with an editable drawing of the graph (but the graph itself is not editable here). On that window, type: 'D' (toggle dynamics on/off), 'L' (toggle labels on/off) and 'F' (fit graph into window) to obtain a nice drawing (the initial one is random). The full list of keyboard commands for the Draw window is displayed when typing 'H' (toggle help message). Besides these keyboard commands, you can use your mouse in obvious ways to edit the drawing.

To quit, type 'S'. The drawing is stored within the graph g and remembered by YAGS in case you want to draw the graph again.

If you are new to GAP, it may be worth mentioning that you need not remember or type all the full names of every YAGS operation: GAP supports command completion. For instance, if you type Path and then hit the <TAB> key, GAP automatically completes the prefix to the unique command that completes it, namely: PathGraph. If, on the other hand, the prefix has several possible completions, then GAP simply beeps, but a second <TAB> makes GAP respond with a list of possible completions, so you can then type some additional keys and perhaps type <TAB> again, and so on.

Example

```
gap> GraphBy<TAB><TAB>
    GraphByAdjMatrix
    GraphByAdjacencies
    GraphByCompleteCover
    GraphByEdges
    GraphByRelation
    GraphByWalks
gap> GraphBy
```

Also, the <UP> and <DOWN> keys are useful to bring back (and perhaps edit) some commands typed earlier in your GAP session. As with any command in GAP/YAGS, in case of doubt, you can always access the online help by typing:

Example

```
gap> ?yags:draw
Help: several entries match this topic - type ?2 to get match [2]

[1] yags: Draw
[2] yags: Drawing
gap> ?1
  B.1-55 Draw

  > Draw( G ) ------------------------------------------- operation

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

  --- many more lines here ---
```

Here, '?' specifies that we want help; 'yags:' specifies on which manual book we want to search (YAGS's book in this case) and 'draw' specifies the topic we would like to be informed about. As it is common, there are more than one place with information on our topic, hence we choose among

the options with '?1' in the next command line. It is not necessary to specify the book, but then you could receive many more options, in different books, about some specific topic.

Now that we know that our graph is the one we want, we can ask YAGS a lot of things about it:

Example

```
gap> Order(g); Size(g); Diameter(g); Girth(g);
5
6
2
3
gap> NumberOfCliques(g); CliqueNumber(g);
4
3
gap> Adjacencies(g);Adjacency(g,4);Adjacency(g,3);
[ [ 2, 4, 5 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3, 5 ], [ 1, 4 ] ]
[ 1, 3, 5 ]
[ 2, 4 ]
gap> VertexDegrees(g);VertexDegree(g,4);VertexDegree(g,3);
[ 3, 2, 2, 3, 2 ]
3
2
gap> IsDiamondFree(g);IsCompleteGraph(g);IsLoopless(g);
true
false
true
gap> Cliques(g);CompletesOfGivenOrder(g,3);
[ [ 1, 4, 5 ], [ 1, 2 ], [ 2, 3 ], [ 3, 4 ] ]
[ [ 1, 4, 5 ] ]
gap> CompletesOfGivenOrder(g,2);
[ [ 1, 2 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ], [ 3, 4 ], [ 4, 5 ] ]
```

Note that in YAGS a *clique* is always *maximal*. This is just a small sample. The full alphabetic list of YAGS operations can be found in Appendix B, and grouped by topic in Appendix A. There is also a one-page pdf file, `cheatsheet-yags.pdf`, which contains a very useful synopsis of many of the most common YAGS operations. See the next section (2.4) for details.

What about *modifying* our graphs? Well, all graphs in YAGS are always immutable, which means that, once created, we can never modify a graph. But we can create new graphs which are variations of existing ones:

Example

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

Note that the graph g remains the same, but the graph h has two additional edges. This is done in this way, because in YAGS everything that is computed about a graph is stored within the graph, so that we never need to compute something twice. This saves time when computing attributes of graphs requiring CPU-intensive algorithms (like computing cliques and clique graphs), but at the expense of having to make a copy of the graph when we just want a small variation of it.

There are a lot of predefined graphs (the full list can be consulted in Appendix A.4):

Example

```
gap> PathGraph(5);CycleGraph(6);CompleteGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size :=
4, Adjacencies := [ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4 ] ] )
Graph( Category := SimpleGraphs, Order := 6, Size :=
6, Adjacencies := [ [ 2, 6 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4, 6 ],
  [ 1, 5 ] ] )
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 ] ] )
gap> CompleteBipartiteGraph(3,3);TreeGraph([2,2,2]);
Graph( Category := SimpleGraphs, Order := 6, Size :=
9, Adjacencies := [ [ 4, 5, 6 ], [ 4, 5, 6 ], [ 4, 5, 6 ],
  [ 1, 2, 3 ], [ 1, 2, 3 ], [ 1, 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> Octahedron;ParapluieGraph;
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 ] ] )
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 ] ] )
```

We have found that `GraphByWalks` (B.7.11) is one of the most useful and versatile ways of specifying graphs:

Example

```
gap> p5:=PathGraph(5);;c6:=CycleGraph(6);;w4:=WheelGraph(4);;
gap> IsIsomorphicGraph(p5,GraphByWalks([1..5]));
true
gap> IsIsomorphicGraph(c6,GraphByWalks([1,2,3,4,5,6,1]));
true
gap> IsIsomorphicGraph(c6,GraphByWalks([1..6],[6,1]));
true
gap> IsIsomorphicGraph(w4,GraphByWalks([1,[2,3,4,5,2]]));
true
gap> sd:=GraphByWalks([1,[2,3,4,5],6],[5,[6,7,8,1],2]);;
gap> IsIsomorphicGraph(SnubDisphenoid,sd);
true
```

YAGS knows about random graphs, so you can take some random graphs and study their parameters. Furthermore, `GraphAttributeStatistics` (B.7.4) can collect statistics on 100 random graphs

at a time returning the collected results of the specified graph parameter on these graphs. The following experiments show, for instance that the values of the minimum degree parameter are much more spread than those of the clique number or those of the diameter.

Example

```
gap> MinDeg:=function(G) return Minimum(VertexDegrees(G)); end;;
gap> g:=RandomGraph(30,1/2);; MinDeg(g); CliqueNumber(g); Diameter(g);
9
6
2
gap> GraphAttributeStatistics(30,1/2,MinDeg);
[ [ 5, 1 ], [ 6, 2 ], [ 7, 6 ], [ 8, 22 ], [ 9, 30 ], [ 10, 30 ],
  [ 11, 5 ], [ 12, 4 ] ]
gap> GraphAttributeStatistics(30,1/2,CliqueNumber);
[ [ 5, 2 ], [ 6, 70 ], [ 7, 24 ], [ 8, 4 ] ]
gap> GraphAttributeStatistics(30,1/2,Diameter);
[ [ 2, 91 ], [ 3, 9 ] ]
```

Finally, it is worth mentioning that algorithms the may take too much time to finish report their progress using the `InfoLevel` mechanism: Enabling and disabling progress reporting is done by changing the `InfoLevel` of `YAGSInfo.InfoClass` to the appropriate level. The default `InfoLevel` is 0. Some of YAGS algorithms report at `InfoLevel` 1, and others at `InfoLevel` 3.

Example

```
gap> SetInfoLevel(YAGSInfo.InfoClass,3);
gap> FullMonoMorphisms(PathGraph(3),CycleGraph(3));
#I [  ]
#I [ 1 ]
#I [ 1, 2 ]
#I [ 1, 3 ]
#I [ 2 ]
#I [ 2, 1 ]
#I [ 2, 3 ]
#I [ 3 ]
#I [ 3, 1 ]
#I [ 3, 2 ]
[  ]
gap> SetInfoLevel(YAGSInfo.InfoClass,0);
gap> FullMonoMorphisms(PathGraph(3),CycleGraph(3));
[  ]
```

This way we can abort the calculation (by typing `Ctr-C`) in case we see that it will take eons to finish. See `YAGSInfo.InfoClass` (B.24.3) for details.

2.4 Cheatsheet

There is a very useful one-page pdf cheatsheet with YAGS's most common functions. It can be consulted in your YAGS installation at `YAGS-DIR/doc/cheatsheet-yags.pdf` or on the web at `http://xamanek.izt.uam.mx/yags/cheatsheet-yags.pdf`. Also, the pdf version of this manual includes it in the next page.

Yags cheatsheet

Graph definitions

Adjacency list

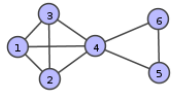
```
g:=GraphByAdjacencies([[],[4],[1,2],[]])
```

Adjacency matrix

```
M:=[[false, true, false], [true, false, true], [false, true, false]];
g:=GraphByAdjMatrix(M);
```

Complete cover

```
g:=GraphByCompleteCover([[1,2,3,4],[4,5,6]]);
```

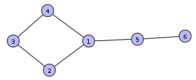


By relation

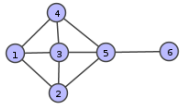
```
f:=function(x,y) return Intersection(x,y)<>[]; end;;
g:=GraphByRelation([[1,2,3],[3,4,5],[5,6,7]],f);
```

By walks

```
g:=GraphByWalks([1,2,3,4,1],[1,5,6]);
```



```
g:=GraphByWalks([1,[2,3,4],5],[5,6]);
```



As intersection graph

```
g:=IntersectionGraph([[1,2,3],[3,4,5],[5,6,7]]);
```

As a copy

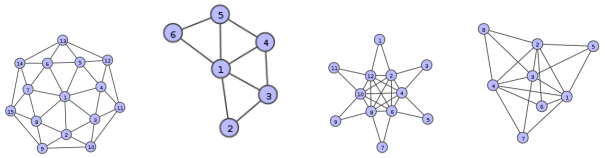
```
h:=CopyGraph(g)
```

As an induced subgraph

```
h:=InducedSubgraph(g,[3,4,6]);
```

Graph families (with parameters)

- `g:=DiscreteGraph(n)`
- `g:=CompleteGraph(n)`
- `g:=PathGraph(n)` n vertices.
- `g:=CycleGraph(n)`
- `g:=CubeGraph(n)`
- `g:=OctahedralGraph(n)`
- `g:=JohnsonGraph(n,r)` Vertices are subsets of $\{1, 2, \ldots, n\}$ with r elements, edges between subsets with intersection of $r - 1$ elements.
- `g:=CompleteBipartiteGraph(n,m)`
- `g:=CompleteMultipartiteGraph(n1,n2[, n3 ...])`
- `g:=WheelGraph(n)`
- `g:=WheelGraph(7,2)` Second optional parameter is the radius of the wheel.
- `g:=FanGraph(4);`
- `g:=SunGraph(6);`
- `g:=SpikyGraph(4);`
- Examples: Wheel, Fan, Sun, Spiky:



Named graphs

Platonic

`Tetrahedron`, `Octahedron`, `Cube`, `Dodecahedron`, `Icosahedron`.

Other

`TrivialGraph`, `DiamondGraph`, `ClawGraph`, `PawGraph`, `HouseGraph`, `BullGraph`, `AntennaGraph`, `KiteGraph`, `SnubDisphenoid`.

Random graphs

- `g:=RandomGraph(n)`
- `g:=RandomGraph(n,p)` Graph with n vertices, each edge with probability p to appear.

Modifying graphs

- `h:=RemoveVertices(g,[1,3]);`
- `h:=AddEdges(g,[[1,2]]);`
- `h:=RemoveEdges(g,[[1,2],[3,4]]);`

Parameters

- `Order(g)`
- `Size(g)`
- `CliqueNumber(g)`
- `VertexDegree(g,v)`

Boolean tests

- `IsCompleteGraph(g)`
- `IsCliqueHelly(g)`
- `IsDiamondFree(g)`

Products

- `p=BoxProduct(g,h)`
- `p=TimesProduct(g,h)`
- `p=BoxTimesProduct(g,h)`
- `p=DisjointUnion(g,h)`
- `p=Join(g,h)`
- `p=GraphSum(g,l)` l is a list of graphs. Suppose that g has n vertices. In the disjoint union of the first n graphs of l (using `TrivialGraphs` if needed to fill n slots), add all edges between graphs corresponding to adjacent vertices in g.
- `p=Composition(g,h)` is the same as `GraphSum(g,l)`, where l is a list of length the order of g, with all components equal to h.

Operators

- `h:=CliqueGraph(g)`
- `h:=CliqueGraph(g,m)` Stops when a maximum of m cliques have been found.
- `h:=LineGraph(g)`
- `h:=ComplementGraph(g)`
- `h:=QuotientGraph(g,p)` p is a partition of vertices. The vertices of h are the parts of p, with two vertices adjacent if there are two vertices adjacent in g in the corresponding parts. Singletons in p may be omitted.
- `h:=QuotientGraph(g,l)` l is a pair of lists of vertices of the same length, with repetitions allowed. In h, each vertex of the first list is identified with the corresponding vertex in the second list.

Lists

- `VertexNames(g)`
- `Cliques(g)`
- `Cliques(g,m)` Stops if a maximum of m cliques have been found.
- `AdjMatrix(g)`
- `Adjaceny(g,v)`
- `Adjacencies(g)`
- `VertexDegrees(g)`
- `Edges(g)`
- `CompletesOfGivenOrder(g,o)`

Distances

- `Distance(g,x,y)`
- `DistanceMatrix(g)`
- `Diameter(g)`
- `Eccentricity(g,x)`
- `Radius(g)`
- `Distances(g,a,b)` a, b are lists of vertices. Returns a list.
- `DistanceSet(g,a,b)` As before, but returns a set.
- `DistanceGraph(g,d)` The graph with vertex set the vertices of g, two vertices adjacent if their distance is in d.
- `PowerGraph(g,n)` Same as the distance graph with set of distance $\{1, \ldots, n\}$.

Chapter 3

Cliques and Clique Graphs

A *clique* is a maximal complete subgraph (other texts use *maxclique* for this concept). It is common to identify induced subgraphs of a graph with their vertex sets; accordingly, a clique in YAGS is actually a set of vertices of a graph such that any two vertices in the clique are adjacent in the considered graph.

The *clique graph*, $K(G)$, of a graph G is the intersection graph of all the cliques of G: Each clique of G is a vertex of $K(G)$, two of them are adjacent in $K(G)$ if and only if they have a non-empty intersection.

A number of YAGS's features concerning cliques and clique graphs are described in this chapter.

3.1 Cliques and Clique Number

You can get the set of all the cliques of a graph by means of `Cliques` (B.3.7); if you want to know the completes of given order (maximal or not) you may use `CompletesOfGivenOrder` (B.3.14) instead.

Example

```
gap> g:=SunGraph(4);;
gap> Cliques(g);
[ [ 2, 4, 6, 8 ], [ 2, 3, 4 ], [ 1, 2, 8 ], [ 4, 5, 6 ], [ 6, 7, 8 ] ]
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 ] ]
```

Note that `CompletesOfGivenOrder` uses a simple, straightforward backtracking algorithm, whereas `Cliques` uses the Bron-Kerbosch algorithm [4] which in our experience is the best algorithm for finding all cliques of a graph in practice. In particular, `Cliques` is much faster than `CompletesOfGivenOrder` (when comparable).

In YAGS all graphs are *immutable*, that is, once created, all graphs always remain exactly the same graph. If you need to modify a graph, you actually construct a new graph by copying the first graph and (for example) adding or deleting some edges (all of this in a single atomic step) therefore creating a new immutable graph. All graph-modifying operations in YAGS (e.g. `AddEdges`, `RemoveEdges`, etc.) work in this way. This is time-consuming if your work involves many frequent graph editions. On the other hand, this design decision allows us to meaningfully store all computed graph attributes within the graph itself: Since the graph is not going to change ever, it will always have the same order, size, clique number, etc. YAGS does exactly this with all graph attributes and properties. This means

that no attribute is ever computed twice for the same graph and in particular it has a very clear effect on computing time:

Example

```
gap> g:=RandomGraph(200);;
gap> Cliques(g);;time;
5784
gap> Cliques(g);;time;
0
```

The *clique number*, $\omega(G)$, is the order of a maximum clique (all cliques are maximal, but they may be of several different orders). On the other hand, the *number of cliques*, is simply the cardinality of the set of all cliques.

Example

```
gap> g:=SunGraph(4);;
gap> CliqueNumber(g);
4
gap> NumberOfCliques(g);
5
```

For random graphs with edge probability p and taking $b := 1/p$, it is known [2] that the distribution of the clique number has a very concentrated peak around $2\log_b n - 2\log_b \log_b n + 2\log_b(\frac{e}{2})$:

Example

```
gap> GraphAttributeStatistics([10,20..70],1/2,CliqueNumber);
[ [ [ 3, 29 ], [ 4, 61 ], [ 5, 9 ], [ 6, 1 ] ],
  [ [ 4, 5 ], [ 5, 64 ], [ 6, 31 ] ],
  [ [ 5, 5 ], [ 6, 64 ], [ 7, 29 ], [ 8, 2 ] ],
  [ [ 6, 20 ], [ 7, 72 ], [ 8, 8 ] ],
  [ [ 7, 55 ], [ 8, 42 ], [ 9, 3 ] ],
  [ [ 7, 17 ], [ 8, 75 ], [ 9, 7 ], [ 10, 1 ] ],
  [ [ 8, 66 ], [ 9, 32 ], [ 10, 2 ] ] ]
gap> List([10,20..70],n->2*Log2(Float(n))
> -2*Log2(Log2(Float(n))) + 2*Log2(FLOAT.E/2));
[ 4.0652, 5.3059, 6.10955, 6.70535, 7.17974, 7.57437, 7.91252 ]
```

Not so much the distribution of the number of cliques:

Example

```
gap> GraphAttributeStatistics([10,20..70],1/2,NumberOfCliques);
[ [ [ 7, 4 ], [ 8, 4 ], [ 9, 18 ], [ 10, 23 ], [ 11, 23 ],
      [ 12, 13 ], [ 13, 8 ], [ 14, 5 ], [ 16, 1 ], [ 17, 1 ] ],
  [ [ 36, 1 ], [ 40, 2 ], [ 41, 3 ], [ 42, 1 ], [ 43, 4 ],
      [ 44, 4 ], [ 45, 2 ], [ 46, 4 ], [ 47, 1 ], [ 48, 5 ],
      [ 49, 2 ], [ 50, 1 ], [ 51, 5 ], [ 52, 6 ], [ 53, 6 ],
      [ 54, 5 ], [ 55, 6 ], [ 56, 4 ], [ 57, 5 ], [ 58, 7 ],
      [ 59, 4 ], [ 60, 2 ], [ 61, 3 ], [ 62, 3 ], [ 63, 4 ],
      [ 64, 3 ], [ 66, 3 ], [ 67, 1 ], [ 68, 1 ], [ 72, 1 ],
      [ 77, 1 ] ],
   [ [ 130, 1 ], [ 133, 1 ], [ 135, 1 ], [ 137, 1 ], [ 142, 1 ],

   --- many more lines here ---

      [ 4626, 1 ], [ 4629, 1 ] ] ]
```

3.2 Clique Graphs

Whenever we have a graph G, we can compute its clique graph $K(G)$, which is the intersection graph of the cliques of G. Much work has been done on clique graphs [28][24][20] and they have even been applied to Loop Quantum Gravity [26][25][27]. It is know that deciding whether a given graph is a clique graph ($G \cong K(H)$ for some H) is NP-complete [1].

Computing the clique graph of a graph is clearly an exponential time operation in the worst case as the maximum number of cliques of a graph of order n is $\alpha \times 3^{\frac{n-\alpha}{3}} = \Theta(3^{\frac{n}{3}})$ [21] (here α it taken such that $\alpha \in \{2,3,4\}$ and $n \equiv \alpha \bmod 3$). However, very often the number of cliques in a graph is much smaller; the following experiment shows that the number of cliques of a graph on 50 vertices is likely between 700 and 1300 instead of the maximum possible which is $2 \times 3^{16} = 86093442$.

Example

```
gap> GraphAttributeStatistics(50,1/2,NumberOfCliques);
[ [ 756, 1 ], [ 762, 1 ], [ 770, 1 ], [ 795, 1 ], [ 826, 2 ],
  [ 832, 1 ], [ 834, 1 ], [ 835, 1 ], [ 856, 1 ], [ 860, 1 ],
  [ 861, 1 ], [ 867, 2 ], [ 870, 1 ], [ 871, 2 ], [ 872, 1 ],
  [ 886, 1 ], [ 887, 1 ], [ 891, 1 ], [ 896, 1 ], [ 897, 2 ],
  [ 898, 1 ], [ 905, 1 ], [ 911, 1 ], [ 916, 2 ], [ 920, 2 ],
  [ 923, 2 ], [ 934, 1 ], [ 938, 1 ], [ 940, 1 ], [ 942, 1 ],
  [ 943, 1 ], [ 944, 1 ], [ 949, 1 ], [ 953, 1 ], [ 963, 1 ],
  [ 965, 1 ], [ 966, 1 ], [ 967, 2 ], [ 970, 1 ], [ 971, 1 ],
  [ 972, 1 ], [ 973, 2 ], [ 975, 1 ], [ 978, 1 ], [ 985, 1 ],
  [ 986, 1 ], [ 988, 1 ], [ 993, 1 ], [ 994, 2 ], [ 997, 1 ],
  [ 998, 1 ], [ 999, 2 ], [ 1002, 1 ], [ 1008, 1 ], [ 1015, 1 ],
  [ 1020, 2 ], [ 1022, 1 ], [ 1025, 1 ], [ 1026, 1 ], [ 1028, 1 ],
  [ 1029, 1 ], [ 1034, 1 ], [ 1047, 1 ], [ 1049, 1 ], [ 1054, 1 ],
  [ 1062, 1 ], [ 1067, 1 ], [ 1069, 1 ], [ 1071, 1 ], [ 1075, 1 ],
  [ 1077, 1 ], [ 1087, 1 ], [ 1088, 1 ], [ 1097, 1 ], [ 1098, 1 ],
  [ 1102, 1 ], [ 1135, 1 ], [ 1139, 1 ], [ 1154, 1 ], [ 1159, 2 ],
  [ 1165, 1 ], [ 1187, 1 ], [ 1191, 1 ], [ 1192, 1 ], [ 1203, 1 ],
  [ 1217, 1 ], [ 1236, 1 ] ]
gap> 2*3^16;
86093442
```

Therefore, we can often compute cliques and clique graphs *in practice*, despite the worst case exponential time. Also, `CliqueGraph` is an attribute of graphs (as most operations in YAGS) and hence, the result is stored within the graph in order to prevent unnecessary recalculation:

Example

```
gap> g:=RandomGraph(80);;
gap> kg:=CliqueGraph(g);;time;
26499
gap> kg2:=CliqueGraph(g);;time;
0
gap> kg2=kg;
true
```

Note that the last line in the previous example is *not* testing for isomorphism, it only tests whether both adjacency matrices are equal. It remains possible, however, that the graph at hand is one of those with a huge number of cliques. We can limit the maximum number of cliques to be computed in `Cliques` and `CliqueGraph` using the optional extra parameter *`maxNumCli`*: With this extra parameter, the computation is aborted when the number of computed cliques reaches *`maxNumCli`*.

Example

```
gap> g:=OctahedralGraph(3);;
gap> CliqueGraph(g,1000);
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> g:=OctahedralGraph(30);; #this has 2^30=1073741824 cliques.
gap> CliqueGraph(g,1000);
fail
```

Alternatively we can use the `InfoLevel` mechanism (B.24.3) to be informed about the progress of clique-related operations in YAGS. This way we can abort the calculation (by typing `Ctr-C`) in case we see that it will take eons to finish.

In YAGS the vertices of a graph are always `[1, 2, ..., Order(G)]`, but often they also have some *names*. This names depend on the way in which the graph is constructed and reflect the origin of the graph. We can get the names of the vertices by using `VertexNames` (B.22.3). In the case of clique graphs, the vertex names are the corresponding cliques of the original graph.

Example

```
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> kg:=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 ] ] )
gap> VertexNames(kg);
[ [ 2, 4, 6, 8 ], [ 2, 3, 4 ], [ 1, 2, 8 ], [ 4, 5, 6 ], [ 6, 7, 8 ] ]
gap> Cliques(g);
[ [ 2, 4, 6, 8 ], [ 2, 3, 4 ], [ 1, 2, 8 ], [ 4, 5, 6 ], [ 6, 7, 8 ] ]
```

Hence, in the previous example, vertex 1 of `kg` is (the one corresponding to) the clique `[ 2, 4, 6, 8 ]` of g, and vertex 2 is `[ 2, 3, 4 ]` etc.

3.3 Basements and Iterated Clique Graphs

Iterated clique graphs are obtained by applying the clique operator several times. As before, we may wonder which vertices of $K^3(g)$ constitute the clique corresponding to some vertex of $K^4(g)$ and this can be settled using `VertexNames` as explained in Section 3.2. But what if we want to know which vertices of g constitute some vertex of $K^4(g)$? This could be done using `VertexNames` at level $K^4(g)$ and then transforming each of the obtained vertices (in $K^3(g)$) using `VertexNames` of $K^3(g)$ and so on... but YAGS already has an operation that does exactly that. The *basement* of a vertex x of an iterated clique graph $K^n(g)$ with respect to some previous iterated clique graph $K^m(g)$ (with $m \leq n$) is, roughly speaking, the set of vertices of $K^m(g)$ that constitute the vertex x, that is, the set of vertices of $K^m(g)$ which are needed for x to exist (see `Basement` (B.2.3) for a formal definition).

Example

```
gap> K:=CliqueGraph;
<Attribute "CliqueGraph">
gap> g:=Icosahedron;;Order(g);
12
gap> kg:=K(g);;Order(kg);
20
gap> k2g:=K(kg);;Order(k2g);
32
gap> k3g:=K(k2g);;Order(k3g);
92
gap> k4g:=K(k3g);;Order(k4g);
472
gap> VertexNames(kg)[1];VertexNames(kg)[3];
[ 1, 2, 3 ]
[ 1, 3, 4 ]
gap> Basement(g,kg,1);Basement(g,kg,3);
[ 1, 2, 3 ]
[ 1, 3, 4 ]
gap> VertexNames(k4g)[2];VertexNames(k4g)[9];
[ 1, 2, 55, 72, 73, 74, 80, 81, 82, 83, 84, 85, 88, 89, 90 ]
[ 1, 2, 3, 4, 10, 16, 81, 82, 85, 87, 88, 89 ]
gap> Basement(k3g,k4g,2);Basement(k3g,k4g,9);
[ 1, 2, 55, 72, 73, 74, 80, 81, 82, 83, 84, 85, 88, 89, 90 ]
[ 1, 2, 3, 4, 10, 16, 81, 82, 85, 87, 88, 89 ]
gap> Basement(k2g,k4g,9);
[ 1, 2, 3, 4, 5, 6, 7, 17, 19, 24, 25, 27, 28, 29, 30, 31 ]
gap> Basement(kg,k4g,9);
[ 1, 2, 3, 4, 5, 7, 8, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 ]
gap> Basement(g,k4g,9);
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ]
```

Basements where introduced (with a different name) in [3] and used again in [23]. They have been useful to study distances and diameters on iterated clique graphs. They are also useful when dealing with stars and neckties.

3.4 Stars and Neckties

Stars and neckties are useful in understanding the structure of iterated clique graphs.

Let G be a graph and $x \in G$. Let us define $x^* := \{q \in K(G) : x \in q\}$. We say that x^* is the *star* of x. Observe that x^* is (induces) a complete subgraph of $K(G)$, and may or may not be a clique of $K(G)$ (i.e. a vertex of $K^2(G)$) depending on whether the star of x is a *maximal* complete subgraph or not.

On the other hand, a vertex in $K^2(G)$ is of the form $Q := \{q_1, q_2, \ldots, q_r\}$ where each q_i is a clique of G (i.e. a vertex of $K(G)$). We say that such *clique of cliques* Q is *a star* if there is some vertex $x \in G$ such that $x \in q_i$ for all $q_i \in Q$; equivalently Q is a star if the total intersection of its cliques is non-empty (i.e. $\cap Q \neq \varnothing$). When Q is not a star (i.e. when $\cap Q = \varnothing$), we say it is a *necktie*. It is easy to show that a clique of cliques Q is a star if and only if its basement at G is exactly the closed neighborhood of a vertex $N[x]$.

Clearly, a star is also the star of some vertex and any star of a vertex which is a clique of cliques is a star. On the other hand, if the star of a vertex x^* is not a clique of cliques, it is surely contained

in some clique of cliques $Q \in K^2(G)$. If for each vertex x we pick such a fixed clique of cliques $*(x) := Q \in K^2(G)$ with $x^* \subseteq Q$, we get the star morphism $* : G \to K^2(G)$.

A very remarkable feature of the second iterated clique graph $K^2(G)$ of G is that the star morphism is often injective and even an isomorphism onto its image.

Example

```
gap> g:=Icosahedron;;
gap> kg:=K(g);;k2g:=K(kg);;
gap> ClosedNeighborhood:=function(g,x)
> return Union(Adjacency(g,x),[x]); end;
function( g, x ) ... end
gap> closNeighs:=List(Vertices(g),x->ClosedNeighborhood(g,x));
[ [ 1 .. 6 ], [ 1, 2, 3, 6, 9, 10 ], [ 1, 2, 3, 4, 10, 11 ],
  [ 1, 3, 4, 5, 7, 11 ], [ 1, 4, 5, 6, 7, 8 ], [ 1, 2, 5, 6, 8, 9 ],
  [ 4, 5, 7, 8, 11, 12 ], [ 5, 6, 7, 8, 9, 12 ],
  [ 2, 6, 8, 9, 10, 12 ], [ 2, 3, 9, 10, 11, 12 ],
  [ 3, 4, 7, 10, 11, 12 ], [ 7 .. 12 ] ]
gap> stars:=Filtered(Vertices(k2g),
> Q->Basement(g,k2g,Q) in closNeighs);
[ 1, 3, 5, 8, 10, 12, 15, 18, 21, 24, 27, 30 ]
gap> h:=InducedSubgraph(k2g,stars);;
gap> IsIsomorphicGraph(g,h);
true
```

Hence $K^2(G)$ is the union of two subgraphs (with some extra edges between them): One composed entirely of stars and the other composed entirely of neckties. The one composed entirely of stars is very similar to G and often even isomorphic to G.

A graph is *clique-Helly* when every family of pairwise intersecting cliques has a non-empty total intersection. Evidently, if G is clique-Helly, then every vertex of $K^2(G)$ is a star. Escalante [6] showed that in the clique-Helly case, $K^2(G)$ is isomorphic to a subgraph of G, namely, the `ParedGraph` (B.16.4) of G (which just removes dominated vertices) and hence the star morphism in the clique-Helly case is an isomorphism exactly when G does not have dominated vertices.

Example

```
gap> g:=BoxTimesProduct(CycleGraph(4),PathGraph(4));;Order(g);
16
gap> IsCliqueHelly(g);
true
gap> k2g:=K(K(g));;pg:=ParedGraph(g);;Order(pg);
8
gap> IsIsomorphicGraph(k2g,pg);
true
gap> k4g:=K(K(k2g));;p2g:=ParedGraph(pg);;Order(p2g);
4
gap> IsIsomorphicGraph(k4g,p2g);
true
gap> DominatedVertices(k4g);
[ ]
gap> IsIsomorphicGraph(k4g,K(K(k4g)));
true
```

3.5 Clique Behavior

When we have a graph G and its iterated clique graphs $K(G), K^2(G), K^3(G), \ldots$ a natural question is: Are all the graphs in the sequence non-isomorphic to each other? The answer is "no" if and only if $K^n(G) \cong K^m(G)$ for some $n \neq m$ if and only if there is a finite bound for the sequence of orders of the iterated clique graphs $|K^n(G)|$. When this happens, we say that G is *clique convergent* and otherwise, we say that it is *clique divergent*. To determine the *clique behavior* of a graph consist in deciding whether it is clique convergent or clique divergent. It is an open problem whether the clique behavior is algorithmically decidable or not [16] and Meidanis even asked if the clique operator has the computing power needed to simulate any Turing Machine, see http://www.ic.unicamp.br/~meidanis/research/clique/ (fetched in 2001), but that does not prevent us from trying to determine clique behavior for specific graphs.

The first thing to try when determining the clique behavior of a graph is simply to iterate the clique operator on it and check for the orders, if we see the order stabilizes, we have a candidate where we can check for isomorphism.

Example

```
gap> g:=TimesProduct(Icosahedron,CompleteGraph(3));;Order(g);
36
gap> g1:=K(g);;Order(g1);
120
gap> g2:=K(g1);;Order(g2);
156
gap> g3:=K(g2);;Order(g3);
120
gap> IsIsomorphicGraph(g1,g3);
true
```

Often however, the orders just keep growing. But then the second thing to try (or even the first!) is to take the `CompletelyParedGraph` (B.3.12) of G since it is known that the clique behavior is invariant under removal of dominated vertices [7]. In the following example `g` is clique convergent because `h` is so, even if a direct calculation of the iterated clique graphs of `g` just ends in memory overflow.

Example

```
gap> cp7:=ComplementGraph(PathGraph(7));;
gap> g:=ComplementGraph(TimesProduct(cp7,cp7));;Order(g);
49
gap> g1:=K(g);;Order(g1);
204
gap> g2:=K(g1);;Order(g2);
7193
gap> g3:=K(g2);;Order(g3);
     --- user interrupt or recursion trap here ---
brk> quit;
gap> h:=CompletelyParedGraph(g);;Order(h);
1
gap> IsIsomorphicGraph(h,K(h));
true
```

It is even advisable construct the combined operator PK (compute the clique graph and then take the completely pared graph) and to compute the sequence of graphs under the iterated application of the PK operator: $(PK)^n(G)$ is clique convergent (for any n) if and only if G is clique convergent.

Example

```
gap> g:=WheelGraph(4,4);;Order(g);
17
gap> g1:=K(g);;Order(g1);
28
gap> g2:=K(g1);;Order(g2);
37
gap> g3:=K(g2);;Order(g3);
60
gap> g4:=K(g3);;Order(g4);
185
gap> g5:=K(g4);;Order(g5);
2868
gap> #too many cliques to continue in this way.
gap> PK:=function(g) return CompletelyParedGraph(K(g)); end;
function( g ) ... end
gap> h1:=PK(g);;Order(h1);
24
gap> h2:=PK(h1);;Order(h2);
25
gap> h3:=PK(h2);;Order(h1);
24
gap> h3:=PK(h2);;Order(h3);
16
gap> h4:=PK(h3);;Order(h4);
13
gap> h5:=PK(h4);;Order(h5);
8
gap> h6:=PK(h5);;Order(h6);
1
gap> IsIsomorphicGraph(h6,K(h6));
true
```

If a graph is clique convergent, it must also be convergent under the PK operator; It is an open problem to determine whether the opposite is also true.

If G is clique convergent, then in principle we can determine that by computer (although there are sometimes insufficient memory or time) but that is not so for clique divergent graphs. Determining clique divergence for graphs can be quite challenging and there is even a graph on eight vertices (the `SnubDisphenoid`) which seems to be divergent but nobody has a proof of it yet [17].

Example

```
gap> g:=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 ] ] )
gap> g1:=K(g);;Order(g1);
12
gap> g2:=K(g1);;Order(g2);
20
gap> g3:=K(g2);;Order(g3);
56
```

```
gap> g4:=K(g3);;Order(g4);
1076
```

The fifth iterated clique graphs of the `SnubDisphenoid` has at least 7.37×10^9 vertices, but it is estimated that the number is more likely around 10^{22}.

How to proceed then? Well there are many studied families of graphs whose clique behavior have been settled (including: the octahedral graphs, cographs, regular locally cyclic graphs (like the Icosahedron and the locally C_6 graphs), some circulants, clockworkgraphs, some comparability graphs, locally colorable graphs) and some techniques that may be applicable (including: retractions, covering maps, expansivity, rank-divergence, dismantlings, local cutpoints).

Two of these techniques have been successful more often than others:

(1) If your graph G has a non-trivial automorphism $f : G \to G$ that sends each vertex outside of its closed neighborhood ($f(x) \notin N[x]$), then there are chances that you can apply the rank-divergence techniques [17][18].

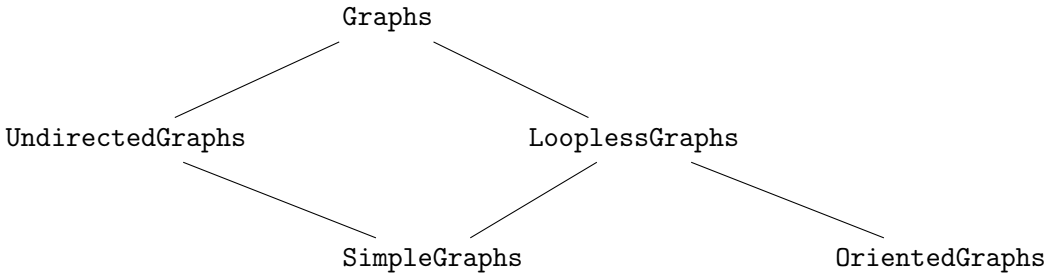
(2) If your graph G is more like a random graph, then there are good chances that it has a retraction to a d-dimensional octahedron with at least 6 vertices, which implies the divergence of G [22]. In our experience, a good way to look up for such a retraction is to find the cliques of G and then try to see if any of the cliques extends in G to an induced octahedral graph (`OctahedralGraph` (B.15.1)) of at least 6 vertices. The existence of an induced octahedral subgraph with one of its faces being a clique of G is sufficient to prove the existence of a retraction from G to the octahedral subgraph [11][19].

We plan to incorporate soon an operation that applies the known techniques for clique divergence and convergence to a given graph in order to try to determine its clique behavior.

Chapter 4

Graph Categories

By default, all graphs in YAGS are simple, i.e. all graphs belong to the `SimpleGraphs` category. There are 5 graph categories in YAGS, namely: `Graphs` (B.7.13), `UndirectedGraphs` (B.21.3), `LooplessGraphs` (B.12.4), `SimpleGraphs` (B.19.3) and `OrientedGraphs` (B.15.5). The inclusion relations among them is as follows:



The most general of these categories is `Graphs`: every graph in YAGS belongs to some category and, by inclusion, every graph belongs to the category `Graphs`. By definition a graph in `Graphs` may contain loops, arrows and edges (which in YAGS are exactly the same as two opposite arrows); another way to say it, is that a graph is anything that can be represented as a binary matrix (its adjacency matrix). In particular, no multiple/parallel edges/arrows are allowed in a graph in YAGS. Likewise, each of the YAGS's graph categories have their own characteristic properties:

`Graphs`	May contain loops, arrows and edges.
`UndirectedGraphs`	Can not contain plain arrows (only edges and loops).
`LooplessGraphs`	Can not contain loops (only arrows and edges).
`SimpleGraphs`	Can not contain loops nor arrows (only edges).
`OrientedGraphs`	Can not contain edges nor loops (only arrows).

Graph categories simplify things for users: for example in the category `SimpleGraphs`, a *complete graph* may be defined as a graph containing "all possible edges" among their vertices, but "all possible edges" in the category `Graphs` includes the loops, while in the category `OrientedGraphs`, it can only contain one arrow for each pair of vertices.

The graph category used for constructing graphs forbids you to add a loop accidentally or to forget to include one of the arrows that constitute an edge in a simple graph: Every graph created in YAGS is forced to comply with its graph category's characteristic properties.

YAGS supports several mechanisms to carefully control the graphs categories used to construct your graphs. These are explained in the following sections.

4.1 The Default Graph Category

The DefaultGraphCategory controls (in the absence of other indications) the graph category to which the new graphs belong. It can not be changed directly as if it were a normal variable, instead, it can be changed by the method SetDefaultGraphCategory (B.19.2)

Example

```
gap> DefaultGraphCategory;
<Category "SimpleGraphs">
gap> SetDefaultGraphCategory(OrientedGraphs);
gap> DefaultGraphCategory;
<Category "OrientedGraphs">
gap> DefaultGraphCategory:=LooplessGraphs;
Error, Variable: 'DefaultGraphCategory' is read only
```

The effect on the constructed graphs is very noticeable: look at the adjacencies of these graphs:

Example

```
gap> SetDefaultGraphCategory(Graphs);CompleteGraph(4);
Graph( Category := Graphs, Order := 4, Size := 16, Adjacencies :=
[ [ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ] ] )
gap> SetDefaultGraphCategory(LooplessGraphs);CompleteGraph(4);
Graph( Category := LooplessGraphs, Order := 4, Size :=
12, Adjacencies := [ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ],
  [ 1, 2, 3 ] ] )
gap> SetDefaultGraphCategory(UndirectedGraphs);CompleteGraph(4);
Graph( Category := UndirectedGraphs, Order := 4, Size :=
10, Adjacencies := [ [ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ],
  [ 1, 2, 3, 4 ] ] )
gap> SetDefaultGraphCategory(OrientedGraphs);CompleteGraph(4);
Graph( Category := OrientedGraphs, Order := 4, Size :=
6, Adjacencies := [ [ 2, 3, 4 ], [ 3, 4 ], [ 4 ], [  ] ] )
gap> SetDefaultGraphCategory(SimpleGraphs);CompleteGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size :=
6, Adjacencies := [ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ],
  [ 1, 2, 3 ] ] )
```

When constructing a graph, YAGS always forces the new graphs to comply with its category, hence, in the case of OrientedGraphs in the previous example, it has to remove one of the arrows conforming the edge for each pair of vertices of the graph. Sometimes it may not be evident which arrow will YAGS choose to remove, but in general, YAGS tries to make sense:

Example

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

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

Therefore, if you always work with `SimpleGraphs`, YAGS defaults are perfect for you. If, in the other hand you always work with `OrientedGraphs` (also known as *digraphs*), you probably would want to start all your sessions by changing the default graph category to that... or even better, you may want to create a startup file that does that automatically every time you start a YAGS session.

On the other hand, your work may involve graphs from more than one graph category. In such a case, it is advisable to continue reading all of this chapter.

4.2 The Target Graph Category

The default graph category is only part of the story. When constructing new graphs from existing ones, it may be natural to construct the new graph in the least common category that contains the original graphs, regardless of the `DefaultGraphCategory`.

For instance, if we have graphs `g` and `h` that belong to the categories of `SimpleGraphs` and `OrientedGraphs` (respectively), then a new graph which is the `BoxTimesProduct` (also known as the *strong product*) of them, should belong to the least common category of both, namely to the `LooplessGraphs` category (see the diagram at the beginning of the chapter). This is what YAGS does:

Example

```
gap> SetDefaultGraphCategory(SimpleGraphs);
gap> g:=PathGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size :=
3, Adjacencies := [ [ 2 ], [ 1, 3 ], [ 2, 4 ], [ 3 ] ] )
gap> SetDefaultGraphCategory(OrientedGraphs);
gap> h:=PathGraph(4);
Graph( Category := OrientedGraphs, Order := 4, Size :=
3, Adjacencies := [ [ 2 ], [ 3 ], [ 4 ], [  ] ] )
gap> SetDefaultGraphCategory(UndirectedGraphs);
gap> s:=BoxTimesProduct(g,h);
Graph( Category := LooplessGraphs, Order := 16, Size :=
54, Adjacencies := [ [ 2, 5, 6 ], [ 3, 6, 7 ], [ 4, 7, 8 ], [ 8 ],
  [ 1, 2, 6, 9, 10 ], [ 2, 3, 7, 10, 11 ], [ 3, 4, 8, 11, 12 ],
  [ 4, 12 ], [ 5, 6, 10, 13, 14 ], [ 6, 7, 11, 14, 15 ],
  [ 7, 8, 12, 15, 16 ], [ 8, 16 ], [ 9, 10, 14 ], [ 10, 11, 15 ],
  [ 11, 12, 16 ], [ 12 ] ] )
gap> s in UndirectedGraphs; s in LooplessGraphs;
false
true
gap> DefaultGraphCategory;
<Category "UndirectedGraphs">
```

Exactly how does YAGS decide this? Well, with very few and evident exceptions (such as `Orientations` (B.15.4)), YAGS's functions that construct graphs, always calls internally the function `TargetGraphCategory` (B.20.1), and passes to it those of the original parameters which are graphs.

`TargetGraphCategory` returns the graph category indicated by GAP's *options stack* if any (see the next section), else it returns the least common category containing all of its parameters if any, or else (if it is called without parameters), `TargetGraphCategory` returns the `DefaultGraphCategory`.

4.3 Changing the Target Graph Category in Place

GAP provides a wonderful feature named *options stack*. Consult GAP's documentation on the topic for a full explanation. For YAGS purposes, the short story is that you may specify the desired graph category directly in the same command used to construct the graph without the need to change the default graph category as in the following example:

Example

```
gap> SetDefaultGraphCategory(Graphs);
gap> DefaultGraphCategory;
<Category "Graphs">
gap> g1:=CompleteGraph(4);
Graph( Category := Graphs, Order := 4, Size := 16, Adjacencies :=
[ [ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ] ] )
gap> DefaultGraphCategory;
<Category "Graphs">
gap> g2:=CompleteGraph(4:GraphCategory:=SimpleGraphs);
Graph( Category := SimpleGraphs, Order := 4, Size :=
6, Adjacencies := [ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ],
  [ 1, 2, 3 ] ] )
gap> DefaultGraphCategory;
<Category "Graphs">
gap> g3:=CompleteGraph(4:GraphCategory:=OrientedGraphs);
Graph( Category := OrientedGraphs, Order := 4, Size :=
6, Adjacencies := [ [ 2, 3, 4 ], [ 3, 4 ], [ 4 ], [  ] ] )
gap> DefaultGraphCategory;
<Category "Graphs">
gap> h:=DisjointUnion(g2,g3);
Graph( Category := LooplessGraphs, Order := 8, Size :=
18, Adjacencies := [ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ],
  [ 1, 2, 3 ], [ 6, 7, 8 ], [ 7, 8 ], [ 8 ], [  ] ] )
gap> DefaultGraphCategory;
<Category "Graphs">
gap> h2:=DisjointUnion(g2,g3:GraphCategory:=UndirectedGraphs);
Graph( Category := UndirectedGraphs, Order := 8, Size :=
12, Adjacencies := [ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ],
  [ 1, 2, 3 ], [ 6, 7, 8 ], [ 5, 7, 8 ], [ 5, 6, 8 ], [ 5, 6, 7 ] ] )
gap> DefaultGraphCategory;
<Category "Graphs">
```

This method of specifying the desired category is useful to copy a graph from one category to another using `CopyGraph` (B.3.20):

Example

```
gap> SetDefaultGraphCategory(SimpleGraphs);
gap> g:=PathGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size :=
```

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

Chapter 5

Morphisms of Graphs

5.1 A Quick Start

A *morphism of graphs* $f : G \to H$ (Also known as *homomorphisms*) is a function from the vertex set of one graph to the vertex set of another, $f : V(G) \to V(H)$, such that some properties (adjacency for instance) are preserved. In YAGS such a function is represented by a list `F=[f(1),f(2),...f(n)]`. For instance, the list `F=[2,3,4,3]` represents a morphism that maps vertex 1 of `G` onto vertex 2 of `H` and also maps 2 to 3, 3 to 4 and 4 to 3. In this example, `F` implicitly says that `G` has exactly 4 vertices and that `H` has at least 4 vertices.

YAGS has a very rich and flexible set of operations to deal with *graph morphisms* which we describe in the next sections. All these operations report progress at `InfoLevel` 3 (see B.24.3 and Section 6.4).

Here we describe only the most used ones. The operations dealing with morphisms are organized in triplets, like the following one:

```
FullMonoMorphism(G,H)
FullMonoMorphisms(G,H)
NextFullMonoMorphism(G,H,F)
```

All three of these operations refer to the same kind of morphisms, f, which are `Morphisms` (the image of an edge is an edge), `Mono` (vertex-injective) and `Full` (i.e. $f(x)f(y) \in E(H) \Rightarrow \exists x'y' \in E(G)$ with $f(x'y') = f(x)f(y)$). The first two operations simply return either *one* or *all* the morphisms between two given graphs:

Example

```
gap> p3:=PathGraph(3);c4:=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> FullMonoMorphism(p3,c4);
[ 1, 2, 3 ]
gap> FullMonoMorphisms(p3,c4);
[ [ 1, 2, 3 ], [ 1, 4, 3 ], [ 2, 1, 4 ], [ 2, 3, 4 ], [ 3, 2, 1 ],
  [ 3, 4, 1 ], [ 4, 1, 2 ], [ 4, 3, 2 ] ]
```

The third operation, NextFullMonoMorphism receives as parameters, besides the two given graphs, a *partial morphism* F. As you may have guessed a partial morphism is any prefix of a morphism, so in case [1, 2, 3] is a morphism, it follows that [1, 2, 3], [1, 2], [1] and [ ] are partial morphisms.

Our operation NextFullMonoMorphism then, returns the next morphism, then one following the given partial morphism F in lexicographic order. It also stores this next morphism in the variable F so we can iteratively call NextFullMonoMorphism to obtain all morphisms one by one:

Example

```
gap> p3:=PathGraph(3);;c4:=CycleGraph(4);;
gap> f:=[3,4];;NextFullMonoMorphism(p3,c4,f);
[ 3, 4, 1 ]
gap> f;
[ 3, 4, 1 ]
gap> NextFullMonoMorphism(p3,c4,f);
[ 4, 1, 2 ]
gap> f;
[ 4, 1, 2 ]
gap> NextFullMonoMorphism(p3,c4,f);
[ 4, 3, 2 ]
gap> NextFullMonoMorphism(p3,c4,f);
fail
gap> f;
[ fail ]
gap> NextFullMonoMorphism(p3,c4,f);
[ 1, 2, 3 ]
gap> NextFullMonoMorphism(p3,c4,f);
[ 1, 4, 3 ]
gap> f:=[];
[ ]
gap> NextFullMonoMorphism(p3,c4,f);
[ 1, 2, 3 ]
gap> NextFullMonoMorphism(p3,c4,f);
[ 1, 4, 3 ]
```

Note that f:=[ ] and f:=[fail] are always considered partial morphisms; these are useful to compute the first morphism and to report when there are no more morphisms to find. Please note that NextFullMonoMorphism will not check whether the given partial morphism is actually a partial morphism. This is done this way for efficiency, since actually both FullMonoMorphism and FullMonoMorphisms are implemented in terms of NextFullMonoMorphism.

NextFullMonoMorphism is useful when the set of all morphisms from G to H is too big: This way, given enough time, we can process all of the morphisms one by one even if the set of all morphisms does not fit in memory.

The reader, may have noticed that these operations are precisely what is needed to implement IsInducedSubgraph:

Example

```
gap> IsInducedSubgraph:=function(h,g)
> return FullMonoMorphism(h,g)<>fail; end;
function( h, g ) ... end
gap> IsInducedSubgraph(PathGraph(3),CycleGraph(4));
true
```

```
gap> IsInducedSubgraph(PathGraph(4),CycleGraph(4));
false
```

If your morphisms of choice are not `Full` nor `Mono`, you can simply use:

```
Morphism(G,H)
Morphisms(G,H)
NextMorphism(G,H,F)
```

just like we did with the previous triplet of operations.

Example

```
gap> Morphism(PathGraph(3),CycleGraph(4));
[ 1, 2, 1 ]
gap> Morphisms(PathGraph(3),CycleGraph(4));
[ [ 1, 2, 1 ], [ 1, 2, 3 ], [ 1, 4, 1 ], [ 1, 4, 3 ], [ 2, 1, 2 ],
  [ 2, 1, 4 ], [ 2, 3, 2 ], [ 2, 3, 4 ], [ 3, 2, 1 ], [ 3, 2, 3 ],
  [ 3, 4, 1 ], [ 3, 4, 3 ], [ 4, 1, 2 ], [ 4, 1, 4 ], [ 4, 3, 2 ],
  [ 4, 3, 4 ] ]
gap> f:=[4,3];NextMorphism(PathGraph(3),CycleGraph(4),f);
[ 4, 3 ]
[ 4, 3, 2 ]
gap> NextMorphism(PathGraph(3),CycleGraph(4),f);
[ 4, 3, 4 ]
gap> NextMorphism(PathGraph(3),CycleGraph(4),f);
fail
gap> NextMorphism(PathGraph(3),CycleGraph(4),f);
[ 1, 2, 1 ]
```

Also, this particular type of morphisms is what we need to implement `IsKColorable`:

Example

```
gap> IsKColorable:=function(g,k)
> return Morphism(g,CompleteGraph(k))<>fail; end;
function( g, k ) ... end
gap> IsKColorable(CycleGraph(6),2);
true
gap> IsKColorable(CycleGraph(5),2);
false
gap> IsKColorable(CycleGraph(5),3);
true
```

The full list of predefined types of morphisms that YAGS knows about is explained in the next section.

5.2 Predefined Types of Morphisms

Following the same organization of operations in triplets as explained in the previous section, we present now the full list of YAGS's operations for predefined types of morphisms. The operations that start with a hash mark (#) are not yet implemented, but they are there as a place holders for a future implementation.

Operations for predefined types of morphisms

```
NextMetricMorphism(G,H,F)
NextEpiMetricMorphism(G,H,F)
NextMonoMorphism(G,H,F)
NextFullMonoMorphism(G,H,F)
NextBiMorphism(G,H,F)
NextFullBiMorphism(G,H,F)
NextCompleteEpiWeakMorphism(G,H,F)
NextCompleteEpiMorphism(G,H,F)
NextCompleteWeakMorphism(G,H,F)
NextCompleteMorphism(G,H,F)
#NextFullEpiWeakMorphism(G,H,F)
#NextFullEpiMorphism(G,H,F)
#NextFullWeakMorphism(G,H,F)
#NextFullMorphism(G,H,F)
NextEpiWeakMorphism(G,H,F)
NextEpiMorphism(G,H,F)
NextWeakMorphism(G,H,F)
NextMorphism(G,H,F)

MetricMorphism(G,H)
EpiMetricMorphism(G,H)
MonoMorphism(G,H)
FullMonoMorphism(G,H)
BiMorphism(G,H)
FullBiMorphism(G,H)
CompleteEpiWeakMorphism(G,H)
CompleteEpiMorphism(G,H)
CompleteWeakMorphism(G,H)
CompleteMorphism(G,H)
#FullEpiWeakMorphism(G,H)
#FullEpiMorphism(G,H)
#FullWeakMorphism(G,H)
#FullMorphism(G,H)
EpiWeakMorphism(G,H)
EpiMorphism(G,H)
WeakMorphism(G,H)
Morphism(G,H)

MetricMorphisms(G,H)
EpiMetricMorphisms(G,H)
MonoMorphisms(G,H)
FullMonoMorphisms(G,H)
BiMorphisms(G,H)
FullBiMorphisms(G,H)
CompleteEpiWeakMorphisms(G,H)
CompleteEpiMorphisms(G,H)
CompleteWeakMorphisms(G,H)
CompleteMorphisms(G,H)
#FullEpiWeakMorphisms(G,H)
#FullEpiMorphisms(G,H)
#FullWeakMorphisms(G,H)
#FullMorphisms(G,H)
```

```
EpiWeakMorphisms(G,H)
EpiMorphisms(G,H)
WeakMorphisms(G,H)
Morphisms(G,H)
```

Here, several name fragments are used in a uniform way:

- `Morphism`: The images of adjacent vertices are adjacent (except with prefix `Weak`).
- `Weak`: Weakens the notion of morphism so that it is allowed that adjacent vertices go to equal ones. That is, a `WeakMorphism` is one where the images of *adjacent-or-equal* vertices are also adjacent-or-equal.
- `Epi`: The morphism is vertex-surjective.
- `Mono`: The morphism is vertex-injective.
- `Bi`: The morphism is vertex-bijective.
- `Full`: $f(x)f(y) \in E(H) \Rightarrow \exists x'y' \in E(G)$ with $f(x'y') = f(x)f(y)$.
- `Complete`: Whenever a pair of vertices `x, y` are mapped onto an edge of H, the pair `[x, y]` is also an edge (of G).
- `Metric`: The image of any pair of vertices are at the same distance from each other as the original pair of vertices.

All meaningful combinations of these name fragments are present in the full list of operations for predefined types of morphisms. But note that some combinations are, by mathematical reasons, necessarily synonyms like `FullBiMorphism = CompleteBiMorphism = MetricBiMorphism`; in such cases, only one of those names is selected for use in YAGS. Note also that a `FullBiMorphism` is most commonly known as an *isomorphism*.

Indeed, YAGS knows about `FullBiMorphisms` and also about `IsoMorphisms`: the former is implemented together with all the other operations listed in this section, using the general schema explained in the next section, while the latter is implemented with different, more efficient, ad-hoc methods. `IsoMorphism` is faster than `FullBiMorphism`, but `FullBiMorphism` is part of a bigger, more flexible schema.

5.3 Main Procedures

All the morphism operations listed in the previous section are implemented in a uniform, semi-automatic way by means of the following triplet of operations, which are explained in their indicated sections of the manual:

`PropertyMorphism` (B.16.9)
`PropertyMorphisms` (B.16.10)
`NextPropertyMorphism` (B.14.2)

In short, the relation of this triplet and the previous ones is best explained by a few examples:

This operation:	Is the same as:
`BiMorphism(G,H)`	`PropertyMorphism(G,H,[CHK_MONO,CHK_EPI,CHK_MORPH])`
`MetricMorphism(G,H)`	`PropertyMorphism(G,H,[CHK_METRIC,CHK_MORPH])`
`CompleteWeakMorphisms(G,H)`	`PropertyMorphisms(G,H,[CHK_CMPLT,CHK_WEAK])`
`NextEpiMorphism(G,H,F)`	`NextPropertyMorphism(G,H,F,[CHK_EPI,CHK_MORPH])`

In the previous table, there are several *predefined property-checking functions*: `CHK_METRIC`, `CHK_CMPLT`, `CHK_MONO`, `CHK_EPI`, `CHK_WEAK` and `CHK_MORPH`. These are functions that receive, two graphs (`G` and `H`) and a partial morphism (`F`) as parameters and they return `true` whenever `F` is a valid (feasible) partial morphism from `G` to `H` satisfying the required property (i.e. metric, complete, mono, etc.); they all return `false` otherwise.

Example

```
gap> CHK_MORPH;
function( g1, g2, morph ) ... end
gap> Print(CHK_MONO,"\n");
function ( g1, g2, morph )
    local  x, y;
    x := Length( morph );
    y := morph[x];
    if y in morph{[ 1 .. x - 1 ]}  then
        return false;
    fi;
    return true;
end
gap> Print(CHK_EPI,"\n");
function ( g1, g2, morph )
    return Order( g2 ) - Length( Set( morph ) )
      <= Order( g1 ) - Length( morph );
end
```

Note that `CHK_MONO` assumes that only the last element in the partial morphism needs to be verified for the sought property. This is correct in general since what `NextPropertyMorphism` does is to continually try to construct a new (longer) partial morphism from an existing one, so the sought property was already checked in all prefixes of the current partial morphism (The precise technique used by `NextPropertyMorphism` is known as backtracking, and it is described in the next chapter).

It is usually required to include at least one of `CHK_WEAK` or `CHK_MORPH` in the list of properties to check used by the `PropertyMorphism` triplet, since otherwise, no adjacency-preserving function is ever verified and then the resulting maps are more properly named "functions" rather than "morphisms":

Example

```
gap> k2:=CompleteGraph(2);;I2:=DiscreteGraph(2);;
gap> PropertyMorphisms(k2,I2,[]);
[ [ 1, 1 ], [ 1, 2 ], [ 2, 1 ], [ 2, 2 ] ]
```

5.4 User-Defined Types of Morphisms

There is nothing special about YAGS predefined property-checking functions and the user may write new ones. For example, if we would like to create a new type of weak morphism restricting the

mapping so that the image of a vertex always has a degree greater than or equal to that of the vertex, then we could write:

Example

```
gap> checkdegree:=function(G,H,f)
> local x,fx;
> x:=Length(f);fx:=f[x];
> return VertexDegree(G,x)<=VertexDegree(H,fx);
> end;
function( G, H, f ) ... end
gap> NextSpecialMorphism:=function(G,H,F)
> return NextPropertyMorphism(G,H,F,[CHK_WEAK,checkdegree]);
> end;
function( G, H, F ) ... end
gap> c4:=CycleGraph(4);;p4:=PathGraph(4);;F:=[];;
gap> NextSpecialMorphism(c4,p4,F);
[ 2, 2, 2, 2 ]
gap> NextSpecialMorphism(c4,p4,F);
[ 2, 2, 2, 3 ]
gap> NextSpecialMorphism(c4,p4,F);
[ 2, 2, 3, 2 ]
gap> NextSpecialMorphism(c4,p4,F);
[ 2, 2, 3, 3 ]
gap> SpecialMorphisms:=function(G,H)
> return PropertyMorphisms(G,H,[CHK_WEAK,checkdegree]);
> end;
function( G, H ) ... end
gap> SpecialMorphisms(c4,p4);
[ [ 2, 2, 2, 2 ], [ 2, 2, 2, 3 ], [ 2, 2, 3, 2 ], [ 2, 2, 3, 3 ],
  [ 2, 3, 2, 2 ], [ 2, 3, 2, 3 ], [ 2, 3, 3, 2 ], [ 2, 3, 3, 3 ],
  [ 3, 2, 2, 2 ], [ 3, 2, 2, 3 ], [ 3, 2, 3, 2 ], [ 3, 2, 3, 3 ],
  [ 3, 3, 2, 2 ], [ 3, 3, 2, 3 ], [ 3, 3, 3, 2 ], [ 3, 3, 3, 3 ] ]
```

Note that the property-checking functions must receive three parameters (namely, two graphs `G,H` and a partial morphism `F`) and it is OK (and better for efficiency), if the function assumes that all prefixes of the current partial morphism, already passed the test.

Since all morphism-related operations in YAGS use `Backtrack` (B.2.1), they all report progress at `InfoLevel` 3 (see B.24.3 and Section 6.4). This is useful to have an idea of how much additional time is needed for the current computation to finish and it is also useful for debugging user-defined property-checking functions.

Chapter 6

Backtracking

Backtracking is an algorithmic technique for *searching in combinatorial spaces*: For instance, when you need to find a particular function (morphism, coloring, etc) subject to some criterion (*iso*morphism, *proper* coloring, etc).

In this chapter we describe the technique and the facilities provided by YAGS to aid in the rapid prototyping of backtracking algorithms. This chapter is written for the non-expert programmer, which is who can benefit the most from these facilities.

While the expert programmers will not have any problem designing their own backtracking algorithms, they can still benefit from YAGS's backtracking facilities since it may still be faster to implement/test/prototype a backtracking algorithm using YAGS's facilities. We recommend the expert programmer to go directly to `Backtrack` (B.2.1) where a minimal non-trivial example (for computing derangements) is shown.

The kind of combinatorial problems that can be addressed by backtracking are those that can be represented by a decision tree: those where we have to make a succession of choices to find a solution. These include problems where we want to find: morphisms, isomorphisms, cages, colorings, cliques, Hamiltonian cycles, walks, paths, subgraphs, and much, much more. Combinatorial problems that can be represented by a decision tree are truly everywhere. YAGS backtracking facilities are provided by means of the operations `BacktrackBag(Opts,Chk,Done,Extra)` and `Backtrack(L,Opts,Chk,Done,Extra)`.

6.1 Simplest Examples: Using `Opts` and `Done`

As a concrete example consider graph colorings: You have a set of colors, say `["red", "blue"]` and a graph, say `g:=PathGraph(3)`, whose vertices are `[1, 2, 3]`. Then a *coloring* is a function $f : \{1,2,3\} \to \{red, blue\}$, which in YAGS would be represented by a List `L:=[ f(1), f(2), f(3)]`. There are (obviously) eight of such colorings, which are easy to list using `BacktrackBag(Opts,Chk,Done,Extra)`:

Example

```
gap> colors:=["red","green"];
[ "red", "green" ]
gap> BacktrackBag(colors,ReturnTrue,3,[]);
[ [ "red", "red", "red" ], [ "red", "red", "green" ],
  [ "red", "green", "red" ], [ "red", "green", "green" ],
  [ "green", "red", "red" ], [ "green", "red", "green" ],
  [ "green", "green", "red" ], [ "green", "green", "green" ] ]
```

```
gap> Length(last);
8
```

In the previous example, only two parameters of `BacktrackBag` are really used, namely: `Opts` (which stands for *options*) and `Done`; when used in this way, `Opts` is simply the codomain of the sought function and `Done` is simply the size of the domain. When working with graph colorings it is common to use numbers instead of actual colors or color names, so we could also write even more compactly:

Example

```
gap> BacktrackBag([0,1],ReturnTrue,3,[]);
[ [ 0, 0, 0 ], [ 0, 0, 1 ], [ 0, 1, 0 ], [ 0, 1, 1 ], [ 1, 0, 0 ],
  [ 1, 0, 1 ], [ 1, 1, 0 ], [ 1, 1, 1 ] ]
```

Sometimes you just want a few solutions and not the whole bag. This is specially true when the bag is too huge (which is often the case since the bag tend to grow exponentially in the size of the domain). For these cases, we have `Backtrack(L,Opts,Chk,Done,Extra)`:

Example

```
gap> L:=[];
[  ]
gap> Backtrack(L,[0,1],ReturnTrue,3,[]);
[ 0, 0, 0 ]
gap> L;
[ 0, 0, 0 ]
gap> Backtrack(L,[0,1],ReturnTrue,3,[]);
[ 0, 0, 1 ]
gap> L;
[ 0, 0, 1 ]
gap> L:=[1,1,0];
[ 1, 1, 0 ]
gap> Backtrack(L,[0,1],ReturnTrue,3,[]);
[ 1, 1, 1 ]
gap> L;
[ 1, 1, 1 ]
gap> Backtrack(L,[0,1],ReturnTrue,3,[]);
fail
gap> L;
[ fail ]
```

`Backtrack(L,Opts,Chk,Done,Extra)` returns one solution at a time and it also stores the solution within `L`, so that `L` can be used as a starting point for the search of the next solution. Usually, `L:=[]` is used for the first search. When Calling `Backtrack(L,Opts,Chk,Done,Extra)`, `L` may also contain a *partial* solution (i.e. a prefix of a solution like `L:=[ 1 ]` or `L:=[ 1, 0 ]`), however `Backtrack` will trust the user on this and it will not check that `L` is indeed a partial solution.

`Backtrack` returns `fail` when no more solutions are available, but for technical reasons, `L` must always be a list, so `L:=[ fail ]` is the final value of `L`.

Now, so far, the graph itself have not been used and the parameters `Chk` and `Extra` have not been explained. Both issues are addressed in the next section.

6.2 Full Examples: Using `Chk` and `Extra`

In Graph Theory we are usually more interested in *proper colorings* than just in colorings. A proper coloring is a coloring in which no two adjacent vertices have the same color. We can easily accommodate the new requirement within our backtracking operations. We are going to use `Chk` to check the fulfillment of the condition at each step in the construction of a solution. Moreover, in order to check the new condition we certainly need the extra information contained in the graph we are coloring. This extra information (the graph) is passed in the `Extra` parameter.

More generally, `Chk` is a user-provided function `Chk(L,Extra)` which receives a partial solution `L` and some extra information `Extra`; it returns `false` whenever it knows that `L` can not be completed to a full solution and it returns `true` otherwise. Note then that our backtracking operations internally call `Chk` several times during the process of constructing each solution: once each time `L` grows in length. In each call to `Chk` it is safe to assume that all proper prefixes of `L` have already been verified and approved by `Chk`.

In our example, `L` contains the color choices already made for the first vertices: `1, 2, ... Length(L)`. It is safe to assume that all but the last choice are already checked to satisfy the properness requirement. The last color choice so far is then the one in `L[Length(L)]` and we have to check (within `Chk`) if it also satisfy the properness requirement. We can do it like this:

Example

```
gap> g:=PathGraph(3);;
gap> chk:=function(L,g)
>        local x,y;
>        if L=[] then return true; fi;
>        x:=Length(L);
>        for y in [1..x-1] do
>           if IsEdge(g,[x,y]) and L[x]=L[y] then
>              return false;
>           fi;
>        od;
>        return true;
>    end;
function( L, g ) ... end
gap> BacktrackBag([0,1],chk,Order(g),g);
[ [ 0, 1, 0 ], [ 1, 0, 1 ] ]
```

Now we get only two solutions, as expected. We emphasize here the fact that `Chk` is internally called by `BacktrackBag` each time `L` grows in length. Therefore (for instance) at some point the partial solution `[ 0, 0 ]` is tried and since it is found unfeasible by `Chk` is is discarded and no other partial solution with that prefix is ever tried. This produces huge reductions in execution time as compared to the (naive) approach of computing all the colorings first and then filtering out those which does not satisfy the properness requirement. In particular we can compute proper colorings for graphs where the naive approach fails:

Example

```
gap> g:=PathGraph(70);;
gap> BacktrackBag([0,1],chk,Order(g),g);
[ [ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
      0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0,
      1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
      0, 1, 0, 1, 0, 1 ],
```

```
  [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0,
    1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
    0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0,
    1, 0, 1, 0, 1, 0 ] ]
```

Of course, we can now compute proper colorings for many other graphs as well:

Example

```
gap> g:=CycleGraph(5);;
gap> BacktrackBag([0,1],chk,Order(g),g);
[ ]
gap> BacktrackBag([0,1,2],chk,Order(g),g);
[ [ 0, 1, 0, 1, 2 ], [ 0, 1, 0, 2, 1 ], [ 0, 1, 2, 0, 1 ],
  [ 0, 1, 2, 0, 2 ], [ 0, 1, 2, 1, 2 ], [ 0, 2, 0, 1, 2 ],
  [ 0, 2, 0, 2, 1 ], [ 0, 2, 1, 0, 1 ], [ 0, 2, 1, 0, 2 ],
  [ 0, 2, 1, 2, 1 ], [ 1, 0, 1, 0, 2 ], [ 1, 0, 1, 2, 0 ],
  [ 1, 0, 2, 0, 2 ], [ 1, 0, 2, 1, 0 ], [ 1, 0, 2, 1, 2 ],
  [ 1, 2, 0, 1, 0 ], [ 1, 2, 0, 1, 2 ], [ 1, 2, 0, 2, 0 ],
  [ 1, 2, 1, 0, 2 ], [ 1, 2, 1, 2, 0 ], [ 2, 0, 1, 0, 1 ],
  [ 2, 0, 1, 2, 0 ], [ 2, 0, 1, 2, 1 ], [ 2, 0, 2, 0, 1 ],
  [ 2, 0, 2, 1, 0 ], [ 2, 1, 0, 1, 0 ], [ 2, 1, 0, 2, 0 ],
  [ 2, 1, 0, 2, 1 ], [ 2, 1, 2, 0, 1 ], [ 2, 1, 2, 1, 0 ] ]
gap> g:=Icosahedron;;
gap> Backtrack([],[0,1,2],chk,Order(g),g);
fail
gap> Backtrack([],[0,1,2,3],chk,Order(g),g);
[ 0, 1, 2, 1, 2, 3, 3, 0, 2, 3, 0, 1 ]
```

6.3 Advanced Examples: When `Opts` and `Done` are functions

Our backtracking operations `BacktrackBag(Opts,Chk,Done,Extra)` and `Backtrack(L,Opts,Chk,Done,Extra)` are even more flexible that shown so far. In our previous examples `Opts` was always a list and `Done` was always an integer. Both can also be functions.

When `Opts` is a function, it receives `L` and `Extra`, and then `Opts(L,Extra)` should return the list of options available to extend the partial solution `L` at that particular stage. This way the list of options can be different at different times which is useful for instance when the union of all possible options is too big or even unbounded.

When `Done` is a function, it also receives `L` and `Extra`, and then `Done(L,Extra)` returns `true` whenever `L` is a full solution and it returns `false` otherwise. This is useful when not all the solutions have the same length. Thus the number `N` we used to put in place of the function `Done(L,Extra)` is equivalent to the function `Done:=function(L,Extra) return Length(L) >= N; end;`.

Also, when a number `N` is used in place of `Done`, an implicit upper bound `Length(L)<= N` is added internally to `Chk`, so it is *imperative* to add such an explicit bound to `Chk` when a function is used for `Done` otherwise the backtracking algorithm will try to find longer and longer solutions without bound or end until the memory of the computer is exhausted.

As an example, assume we want to find all the walks on 5-cycle that start at vertex 1 and end at vertex 2 with length at most 4 (at most 5 vertices). Then we can proceed as follows:

Example

```
gap> g:=CycleGraph(5);
Graph( Category := SimpleGraphs, Order := 5, Size :=
5, Adjacencies := [ [ 2, 5 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 1, 4 ]
 ] )
gap> opts:=function(L,g)
>             if L=[] then
>                return [1];
>             else
>                return Adjacency(g,L[Length(L)]);
>            fi;
>    end;
function( L, g ) ... end
gap> chk:=function(L,g) return Length(L)<= 5; end;
function( L, g ) ... end
gap> done:=function(L,g) return L[Length(L)]=2; end;
function( L, g ) ... end
gap> BacktrackBag(opts,chk,done,g);
[ [ 1, 2 ], [ 1, 2, 1, 2 ], [ 1, 2, 3, 2 ], [ 1, 5, 1, 2 ],
  [ 1, 5, 4, 3, 2 ] ]
```

Finally you may wonder why only *one* extra parameter `Extra` is allowed, what if I need more than one? Well, the parameter `Extra` may be any object supported by GAP; indeed YAGS only uses it to pass information to the user-defined functions `Opts(L,Extra)`, `Chk(L,Extra)` and `Done(L,Extra)`. Hence, if you need pass more than one extra parameter, say two graphs `g` and `h`, you just put them in a list (or record, etc) and pass the parameter `Extra:=[g,h]`.

6.4 Debugging Backtracking Algorithms.

Sooner or later you will need to debug a backtracking algorithm that is not working as expected, or at least, you would like to know how much work your algorithm has done and how much remains to be done to decide if it is worth waiting for an answer (since backtracking techniques easily produce algorithms which may take eons to finish).

All of YAGS's backtracking operations report progress info at `InfoLevel` 3 to YAGS's info class `YAGSInfo.InfoClass` (see B.24.3). In short, this means that setting `SetInfoLevel(YAGSInfo.InfoClass,3);` will cause all backtracking operations to report progress information to the console: The contents of `L` will be reported each time it grows in length. Revert to the default behavior by setting the `InfoLevel` to 0 using `SetInfoLevel(YAGSInfo.InfoClass,0);`

Example

```
gap> SetInfoLevel(YAGSInfo.InfoClass,3);
gap> BacktrackBag([0,1],ReturnTrue,3,[]);
#I [  ]
#I [ 0 ]
#I [ 0, 0 ]
#I [ 0, 0, 0 ]
#I [ 0, 0, 1 ]
#I [ 0, 1 ]
#I [ 0, 1, 0 ]
#I [ 0, 1, 1 ]
```

```
#I [ 1 ]
#I [ 1, 0 ]
#I [ 1, 0, 0 ]
#I [ 1, 0, 1 ]
#I [ 1, 1 ]
#I [ 1, 1, 0 ]
#I [ 1, 1, 1 ]
[ [ 0, 0, 0 ], [ 0, 0, 1 ], [ 0, 1, 0 ], [ 0, 1, 1 ], [ 1, 0, 0 ],
[ 1, 0, 1 ], [ 1, 1, 0 ], [ 1, 1, 1 ] ]
gap> SetInfoLevel(YAGSInfo.InfoClass,0);
gap> BacktrackBag([0,1],ReturnTrue,3,[]);
[ [ 0, 0, 0 ], [ 0, 0, 1 ], [ 0, 1, 0 ], [ 0, 1, 1 ], [ 1, 0, 0 ],
  [ 1, 0, 1 ], [ 1, 1, 0 ], [ 1, 1, 1 ] ]
```

The output of the progress info may be redirected to a file or character device by setting the variable `YAGSInfo.InfoOutput` (B.24.4) accordingly.

The information in this section about progress reporting applies to all YAGS functions that internally use `Backtrack` (B.2.1) or `BacktrackBag` (B.2.2), namely `CompletesOfGivenOrder` (B.3.14), `Orientations` (B.15.4) and all morphism-related operations in Chapter 5.

Appendix A

YAGS Functions by Topic

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

A.1 Most Common Functions

- `AddEdges( G, E )`
  Returns a new graph obtained from `G` by adding the list of edges in `E`. (B.1.1)
- `Adjacency( G, x )`
  Returns the list of vertices in `G` which are adjacent to vertex `x`. (B.1.4)
- `AutomorphismGroup( G )`
  Returns the automorphism group of graph `G`. A synonym is `AutGroupGraph( G )`. (B.1.8)
- `BoxProduct( G, H );`
  Returns the box product (or Cartesian product) $G \square H$ of graphs `G` and `H`. (B.2.5)
- `BoxTimesProduct( G, H )`
  Returns the boxtimes product (or strong product) $G \boxtimes H$ of graphs `G` and `H`. (B.2.6)
- `Circulant( n, Jumps )`
  Returns the minimal graph invariant under the cyclic permutation $(1\ 2\ \cdots\ n)$ such that the vertex 1 is adjacent to the vertices in `Jumps`. (B.3.3)
- `CliqueGraph( G )`
  `CliqueGraph( G, maxNumCli )`
  Returns the intersection graph of the (maximal) cliques of `G`; aborts if `maxNumCli` cliques are found. (B.3.5)
- `Cliques( G )`
  `Cliques( G, maxNumCli )`
  Returns the list of (maximal) cliques of `G`; aborts if `maxNumCli` cliques are found. (B.3.7)
- `ComplementGraph( G )`
  Returns a new graph H such that $V(H) = V(G)$ and $xy \in E(H) \iff xy \notin E(G)$. (B.3.9)
- `CompleteGraph( n )`
  Returns the graph on `n` vertices having all possible edges present. (B.3.11)

- `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. (B.3.13)

- `ConnectedComponents( G )`
  Returns the equivalence partition of $V(G)$ corresponding to the equivalence relation "reachable" in `G`. (B.3.17)

- `CycleGraph( n )`
  Returns the cyclic graph on `n` vertices. (B.3.23)

- `Diameter( G )`
  Returns the maximum among the distances between pairs of vertices of `G`. (B.4.3)

- `DiscreteGraph( n )`
  Returns the graph on `n` vertices with no edges. (B.4.5)

- `DisjointUnion( G, H )`
  Returns the disjoint union of two graphs `G` and `H`. (B.4.6)

- `Distance( G, x, y )`
  Returns the minimum length of a path connecting `x` to `y` in `G`. (B.4.7)

- `Draw( G )`
  Draws the graph `G` on a new window. (B.4.15)

- `Edges( G )`
  Returns the list of edges of graph `G`. (B.5.3)

- `GraphAttributeStatistics( OrderList , ProbList , Attribute )`
  Returns statistics for graph attribute `Attribute`. (B.7.4)

- `GraphByAdjacencies( AdjList )`
  Returns a new graph having `AdjList` as its list of adjacencies. (B.7.6)

- `GraphByAdjMatrix( Mat )`
  Returns a new graph created from an adjacency matrix `Mat`. (B.7.7)

- `GraphByCompleteCover( Cover )`
  Returns the minimal graph where the elements of `Cover` are (the vertex sets of) complete subgraphs. (B.7.8)

- `GraphByEdges( L )`
  Returns the minimal graph such that the pairs in `L` are edges. (B.7.9)

- `GraphByRelation( V, Rel )`
  `GraphByRelation( n, Rel )`
  Returns a new graph G where $xy \in E(G)$ iff `Rel`$(x,y) =$`true`. (B.7.10)

- `GraphByWalks( Walk1 , Walk2 ,...)`
  Returns the minimal graph such that `Walk1`, `Walk2`, etc are walks. (B.7.11)

- `GraphSum( G, L )`
  Returns the lexicographic sum of a list of graphs `L` over a graph `G`. (B.7.15)

- `InducedSubgraph( G, V )`
  Returns the subgraph `G` induced by the vertex set `V`. (B.9.4)

- `InNeigh( G, x )`
  Returns the list of in-neighbors of `x` in `G`. (B.9.5)

- `IntersectionGraph( L )`
  Returns the graph G where $V(G) = L$ and $XY \in E(G) \iff X \cap Y \neq \varnothing$. (B.9.7)

- `IsEdge( G, x, y )`
  `IsEdge( G, [x,y] )`
  Returns `true` if `[x,y]` is an edge of `G`. (B.9.15)

- `IsIsomorphicGraph( G, H )`
  Returns `true` when `G` is isomorphic to `H` and `false` otherwise. (B.9.16)

- `Join( G, H )`
  Returns the disjoint union of `G` and `H` with all the possible edges between `G` and `H` added. (B.10.2)

- `LineGraph( G )`
  Returns the intersection graph of the edges of `G`. (B.12.1)

- `Link( G, x )`
  Returns the subgraph induced in `G` by the neighbors of `x`. (B.12.2)

- `MaxDegree( G )`
  Returns the maximum degree in graph `G`. (B.13.1)

- `MinDegree( G )`
  Returns the minimum degree in graph `G`. (B.13.2)

- `Order( G )`
  Returns the number of vertices, of graph `G`. (B.15.3)

- `PathGraph( n )`
  Returns the path graph on `n` vertices. (B.16.5)

- `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. (B.17.3)

- `RandomGraph( n, p )`
  `RandomGraph( n )`
  Returns a random graph of order `n` with edge probability p (a rational in $[0,1]$). (B.18.3)

- `RemoveEdges( G, E )`
  Returns a new graph created from graph `G` by removing the edges in list `E`. (B.18.7)

- `SetDefaultGraphCategory( Catgy )`
  Sets the default graph category to *Catgy*. (B.19.2)

- `Size( G )`
  Returns the number of edges of graph *G*. (B.19.4)

- `TimesProduct( G, H )`
  Returns the times product (or tensor product) $G \times H$ of two graphs *G* and *H*. (B.20.4)

- `TrivialGraph`
  The one vertex graph. (B.20.7)

- `VertexDegree( G, x )`
  Returns the degree of vertex *x* in graph *G*. (B.22.1)

- `VertexNames( G )`
  Returns the list of names of the vertices of *G*. (B.22.3)

- `WheelGraph( n )`
  `WheelGraph( n, r )`
  This is the cone of an *n*-cycle; when present, *r* is the radius of the wheel. (B.23.1)

A.2 Drawing

- `Coordinates( G )`
  Returns the list of coordinates of the vertices of *G* if they exist; fail otherwise. (B.3.19)

- `Draw( G )`
  Draws the graph *G* on a new window. (B.4.15)

- `GraphToRaw( FileName, G )`
  Writes the graph *G* in raw format to the file *FileName*. (B.7.16)

- `GraphUpdateFromRaw( FileName, G )`
  Updates the coordinates of *G* from a file *FileName* in raw format. (B.7.17)

- `SetCoordinates( G, Coord )`
  Sets the coordinates of the vertices of *G*, which are used to draw *G* by `Draw( G )`. (B.19.1)

A.3 Constructing Graphs

- `AddEdges( G, E )`
  Returns a new graph obtained from *G* by adding the list of edges in *E*. (B.1.1)

- `AddVerticesByAdjacencies( G, NewAdjList )`
  Returns a new graph obtained from *G* by adding some vertices with adjacencies described by *NewAdjList*. (B.1.2)

- `Graph( Rec )`
  Returns a new graph created from the information in record *Rec*. (B.7.3)

- `GraphByAdjacencies( AdjList )`
  Returns a new graph having `AdjList` as its list of adjacencies. (B.7.6)

- `GraphByAdjMatrix( Mat )`
  Returns a new graph created from an adjacency matrix `Mat`. (B.7.7)

- `GraphByCompleteCover( Cover )`
  Returns the minimal graph where the elements of `Cover` are (the vertex sets of) complete subgraphs. (B.7.8)

- `GraphByEdges( L )`
  Returns the minimal graph such that the pairs in `L` are edges. (B.7.9)

- `GraphByRelation( V, Rel )`
  `GraphByRelation( n, Rel )`
  Returns a new graph G where $xy \in E(G)$ iff `Rel`$(x,y) =$`true`. (B.7.10)

- `GraphByWalks( Walk1, Walk2,...)`
  Returns the minimal graph such that `Walk1`, `Walk2`, etc are walks. (B.7.11)

- `InducedSubgraph( G, V )`
  Returns the subgraph of `G` induced by the vertex set `V`. (B.9.4)

- `IntersectionGraph( L )`
  Returns the graph G where $V(G) = L$ and $XY \in E(G) \iff X \cap Y \neq \varnothing$. (B.9.7)

- `RandomGraph( n, p )`
  `RandomGraph( n )`
  Returns a random graph of order `n` with edge probability p (a rational in $[0,1]$). (B.18.3)

- `RemoveEdges( G, E )`
  Returns a new graph created from graph `G` by removing the edges in list `E`. (B.18.7)

- `RemoveVertices( G, V )`
  Returns a new graph created from graph `G` by removing the vertices in list `V`. (B.18.8)

A.4 Families of Graphs

- `AGraph`
  A 4-cycle with two pendant vertices on consecutive vertices of the cycle. (B.1.6)

- `AntennaGraph`
  A `HouseGraph` with a pendant vertex (antenna) on the roof. (B.1.7)

- `BullGraph`
  A triangle with two pendant vertices (horns). (B.2.7)

- `ChairGraph`
  A tree with degree sequence 3,2,1,1,1. (B.3.2)

- `Circulant( n, Jumps )`
  Returns the minimal graph invariant under the cyclic permutation $(1\ 2\ \cdots\ n)$ such that the vertex 1 is adjacent to the vertices in *Jumps*. (B.3.3)

- `ClawGraph`
  The graph on 4 vertices, 3 edges, and maximum degree 3. (B.3.4)

- `ClockworkGraph( NNFSList )`
  `ClockworkGraph( NNFSList, rank )`
  `ClockworkGraph( NNFSList, Perm )`
  `ClockworkGraph( NNFSList, rank, Perm )`
  Returns the clockwork graph specified by its parameters. (B.3.8)

- `CompleteBipartiteGraph( n, m )`
  Returns the minimal graph such that all the vertices in $\{1,2,\ldots,n\}$ are adjacent to all the vertices in $\{n+1,n+2,\ldots,n+m\}$. (B.3.10)

- `CompleteGraph( n )`
  Returns the graph on *n* vertices having all possible edges present. (B.3.11)

- `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. (B.3.13)

- `Cube`
  The 1-skeleton of Plato's cube. (B.3.21)

- `CubeGraph( n )`
  Returns the underlying graph of the *n*-hypercube. (B.3.22)

- `CycleGraph( n )`
  Returns the cyclic graph on *n* vertices. (B.3.23)

- `CylinderGraph( b, h )`
  Returns a graph on $b(h+1)$ vertices which is a $\{4,6\}$-regular triangulation of the cylinder. (B.3.24)

- `DartGraph`
  A diamond with a pendant vertex and maximum degree 4. (B.4.1)

- `DiamondGraph`
  The graph on 4 vertices and 5 edges. (B.4.4)

- `DiscreteGraph( n )`
  Returns the graph on *n* vertices with no edges. (B.4.5)

- `Dodecahedron`
  The 1-skeleton of Plato's Dodecahedron. (B.4.12)

- `DominoGraph`
  Two squares glued by an edge. (B.4.14)

- `FanGraph( n )`
  Returns the n-Fan: The join of a vertex and a $(n+1)$-path. (B.6.1)
- `FishGraph`
  A square and a triangle glued by a vertex. (B.6.2)
- `GemGraph`
  The 3-Fan graph. (B.7.1)
- `HouseGraph`
  A 4-Cycle and a triangle glued by an edge. (B.8.2)
- `Icosahedron`
  The 1-skeleton of Plato's icosahedron. (B.9.1)
- `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. (B.10.1)
- `KiteGraph`
  A diamond with a pendant vertex and maximum degree 3. (B.11.1)
- `OctahedralGraph( n )`
  Returns the $(2n-2)$-regular graph on $2n$ vertices. (B.15.1)
- `Octahedron`
  The 1-skeleton of Plato's octahedron. (B.15.2)
- `ParachuteGraph`
  Returns the suspension of a 4-path with a pendant vertex attached to the south pole. (B.16.2)
- `ParapluieGraph`
  A 3-Fan graph with a 3-path attached to the universal vertex. (B.16.3)
- `PathGraph( n )`
  Returns the path graph on n vertices. (B.16.5)
- `PawGraph`
  A triangle with a pendant vertex. (B.16.6)
- `PetersenGraph`
  The 3-regular graph on 10 vertices having girth 5. (B.16.7)
- `RandomCirculant( n )`
  `RandomCirculant( n, k )`
  `RandomCirculant( n, p )`
  Returns a circulant on n vertices with its *jumps* selected randomly. (B.18.2)
- `RGraph`
  A square with two pendant vertices attached to the same vertex of the square. (B.18.9)
- `SnubDisphenoid`
  The 1-skeleton of the 84th Johnson solid. (B.19.5)

- `SpikyGraph( n )`
  Returns a complete on `n` vertices, with an additional complete on `n` vertices glued to each of its (`n`-1)-dimensional faces. (B.19.8)

- `SunGraph( n )`
  Returns a complete graph on `n` vertices with a zigzagging corona of 2`n` vertices glued to a `n`-cycle of the complete graph. (B.19.9)

- `Tetrahedron`
  The 1-skeleton of Plato's tetrahedron. (B.20.2)

- `TorusGraph( n , m )`
  Returns (the underlying graph of) a triangulation of the torus on `nm` vertices. (B.20.5)

- `TreeGraph( arity , depth )`
  `TreeGraph( ArityList )`
  Returns the tree, the connected cycle-free graph, specified by it parameters. (B.20.6)

- `TrivialGraph`
  The one vertex graph. (B.20.7)

- `WheelGraph( n )`
  `WheelGraph( n , r )`
  This is the cone of an `n`-cycle; when present `r` is the radius of the wheel. (B.23.1)

A.5 Small Graphs

- `ConnectedGraphsOfGivenOrder( n )`
  Returns the list of all connected graphs of order `n` (up to isomorphism). (B.3.18)

- `Graph6ToGraph( String )`
  Returns the graph represented by `String` which is encoded using Brendan McKay's graph6 format. (B.7.5)

- `GraphsOfGivenOrder( n )`
  Returns the list of all graphs of order `n` (up to isomorphism). (B.7.14)

- `HararyToMcKay( Spec )`
  Returns the McKay's `index` of a Harary's graph specification `Spec`. (B.8.1)

- `ImportGraph6( Filename )`
  Returns the list of graphs represented in `Filename` which are encoded using Brendan McKay's graph6 format. (B.9.2)

- `McKayToHarary( index )`
  Returns the Harary's graph specification of a McKay's `index`. (B.8.1)

A.6 Attributes and Parameters

- `Adjacencies( G )`
  Returns the list of adjacencies of `G`: The neighbors of vertex `x` are listed in position `x` of that list. (B.1.3)

- `Adjacency( G, x )`
  Returns the list of vertices in `G` which are adjacent to vertex `x`. (B.1.4)

- `AdjMatrix( G )`
  Returns the adjacency matrix of `G`. (B.1.5)

- `AutomorphismGroup( G )`
  Returns the automorphism group of graph `G`. A synonym is `AutGroupGraph( G )`. (B.1.8)

- `BoundaryVertices( G )`
  Returns the list of vertices of `G` that have links isomorphic to a path. But it returns `fail` if `G` is not (the underlying graph of a triangulation of) a compact surface. (B.2.4)

- `ConnectedComponents( G )`
  Returns the equivalence partition of $V(G)$ corresponding to the equivalence relation "reachable" in `G`. (B.3.17)

- `Diameter( G )`
  Returns the maximum among the distances between pairs of vertices of `G`. (B.4.3)

- `Distance( G, x, y )`
  Returns the minimum length of a path connecting `x` to `y` in `G`. (B.4.7)

- `DistanceMatrix( G )`
  Returns an $n \times n$ matrix `D`, where `D[x][y]` is the distance between `x` and `y` in `G`. (B.4.10)

- `Distances( G, A, B )`
  Returns the list of distances between pairs of vertices in $A \times B$. (B.4.8)

- `DistanceSet( G, A, B )`
  Returns the set of distances between pairs of vertices in $A \times B$. (B.4.11)

- `DominatedVertices( G )`
  Returns the set of dominated vertices of `G`. (B.4.13)

- `Eccentricity( G, x )`
  Returns the distance from a vertex `x` in graph `G` to its most distant vertex in `G`. (B.5.2)

- `Edges( G )`
  Returns the list of edges of graph `G`. (B.5.3)

- `Girth( G )`
  Returns the length of a minimum induced cycle in `G`. (B.7.2)

- `GraphAttributeStatistics( OrderList, ProbList, Attribute )`
  Returns statistics for graph attribute `Attribute`. (B.7.4)

- `InteriorVertices( G )`
  Returns the list of vertices of `G` that have links isomorphic to a cycle. But it returns `fail` if `G` is not a compact surface. (B.9.6)

- `IsCompactSurface( G )`
  Returns `true` if every link of `G` is either an `n`-cycle, for $n \geq 4$ or an `m`-path, for $m \geq 2$; it returns `false` otherwise. (B.9.11)

- `IsDiamondFree( G )`
  Returns `true` if `G` is free from induced diamonds, `false` otherwise. (B.9.14)

- `IsEdge( G, x, y )`
  `IsEdge( G, [x,y] )`
  Returns `true` if `[x,y]` is an edge of `G`. (B.9.15)

- `IsLocallyConstant( G )`
  Returns `true` if all the links of `G` are isomorphic to each other; `false` otherwise (B.9.17)

- `IsLocallyH( G, H )`
  Returns `true` if all the links of `G` are isomorphic to `H`; `false` otherwise. (B.9.18)

- `IsLoopless( G )`
  Returns `true` when `G` does not have loops: edges of the form `[x,x]`. (B.9.19)

- `IsOriented( G )`
  Returns `true` if whenever `[x,y]` is an edge (arrow) of `G`, `[y,x]` is not. (B.9.22)

- `IsSimple( G )`
  Returns `true` if `G` contains no loops and no arrows. (B.9.23)

- `IsSurface( G )`
  Returns `true` if every link of `G` is an `n`-cycle, for `n`≥ 4; `false` otherwise. (B.9.24)

- `IsUndirected( G )`
  Returns `true` if, whenever `[x,y]` is an edge (arrow) of `G`, `[y,x]` is also an edge of `G`. (B.9.27)

- `Link( G, x )`
  Returns the subgraph induced in `G` by the neighbors of `x`. (B.12.2)

- `Links( G )`
  Returns the list of subgraphs of `G` induced by the neighbors of each vertex of `G`. (B.12.3)

- `MaxDegree( G )`
  Returns the maximum degree in graph `G`. (B.13.1)

- `MinDegree( G )`
  Returns the minimum degree in graph `G`. (B.13.2)

- `NumberOfConnectedComponents( G )`
  Returns the number of connected components of `G`. (B.14.4)

- `Order( G )`
  Returns the number of vertices, of graph `G`. (B.15.3)

- `Radius( G )`
  Returns the minimal eccentricity among the vertices of graph `G`. (B.18.1)
- `Size( G )`
  Returns the number of edges of graph `G`. (B.19.4)
- `SpanningForest( G )`
  Returns a spanning forest of `G`. (B.19.6)
- `SpanningForestEdges( G )`
  Returns the edges of a spanning forest of `G`. (B.19.7)
- `VertexDegree( G, x )`
  Returns the degree of vertex `x` in graph `G`. (B.22.1)
- `VertexDegrees( G )`
  Returns the list of degrees of the vertices in graph `G`. (B.22.2)
- `VertexNames( G )`
  Returns the list of names of the vertices of `G`. (B.22.3)
- `Vertices( G )`
  Returns the list `[1..Order( G )]`. (B.22.4)

A.7 Unary Operators

- `CliqueGraph( G )`
  `CliqueGraph( G, maxNumCli )`
  Returns the intersection graph of the (maximal) cliques of `G`; aborts if `maxNumCli` cliques are found. (B.3.5)
- `ComplementGraph( G )`
  Returns a new graph H such that $V(H) = V(G)$ and $xy \in E(H) \iff xy \notin E(G)$. (B.3.9)
- `CompletelyParedGraph( G )`
  Returns the graph obtained from `G` by iteratively removing all dominated vertices. (B.3.12)
- `Cone( G )`
  Returns a new graph obtained from `G` by adding a new vertex which is adjacent to all vertices of `G`. (B.3.16)
- `DistanceGraph( G, Dist )`
  Returns a new graph with the same vertices as `G`, where two vertices are adjacent iff the distance between them in `G` belongs to `Dist`. (B.4.9)
- `InducedSubgraph( G, V )`
  Returns the subgraph of graph `G` induced by the vertex set `V`. (B.9.4)
- `LineGraph( G )`
  Returns the intersection graph of the edges of `G`. (B.12.1)

- `ParedGraph( G )`
  Returns the induced subgraph obtained from `G` by removing its dominated vertices. (B.16.4)

- `PowerGraph( G, exp )`
  Returns a new graph where two vertices are neighbors iff their distance in `G` is less than or equal to `exp`. (B.16.8)

- `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. (B.17.3)

- `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. (B.19.10)

A.8 Binary Operators

- `BoxProduct( G, H );`
  Returns the box product (or Cartesian product) $G \square H$ of graphs `G` and `H`. (B.2.5)

- `BoxTimesProduct( G, H )`
  Returns the boxtimes product (or strong product) $G \boxtimes H$ of graphs `G` and `H`. (B.2.6)

- `Composition( G, H )`
  Returns the composition $G[H]$ of two graphs `G` and `H`. (B.3.15)

- `DisjointUnion( G, H )`
  Returns the disjoint union of two graphs `G` and `H`. (B.4.6)

- `GraphSum( G, L )`
  Returns the lexicographic sum of a list of graphs `L` over a graph `G`. (B.7.15)

- `Join( G, H )`
  Returns the disjoint union of `G` and `H` with all the possible edges between `G` and `H` added. (B.10.2)

- `TimesProduct( G, H )`
  Returns the times product (or tensor product) $G \times H$ of two graphs `G` and `H`. (B.20.4)

A.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`. (B.2.3)

- `CliqueGraph( G )`
  `CliqueGraph( G, maxNumCli )`

Returns the intersection graph of the (maximal) cliques of `G`; aborts if `maxNumCli` cliques are found. (B.3.5)

- `CliqueNumber( G )`
  Returns the order, $\omega(G)$, of a maximum clique of `G`. (B.3.6)
- `Cliques( G )`
  `Cliques( G, maxNumCli )`
  Returns the list of (maximal) cliques of `G`; aborts if `maxNumCli` cliques are found. (B.3.7)
- `CompletesOfGivenOrder( G, ord )`
  Returns the list of vertex sets of all complete subgraphs of order `ord` of `G`. (B.3.14)
- `IsCliqueGated( G )`
  Returns `true` if `G` is a clique gated graph. (B.9.9)
- `IsCliqueHelly( G )`
  Returns `true` if the set of (maximal) cliques of `G` satisfy the *Helly* property. (B.9.10)
- `IsComplete( G, L )`
  Returns `true` if `L` induces a complete subgraph of `G`. (B.9.12)
- `IsCompleteGraph( G )`
  Returns `true` if graph `G` is a complete graph, `false` otherwise. (B.9.13)
- `NumberOfCliques( G )`
  `NumberOfCliques( G, maxNumCli )`
  Returns the number of (maximal) cliques of `G`. (B.14.3)

A.10 Morphisms of Graphs

We list here only primitive operations, many derived operations (over forty) for morphisms of graphs are discussed in Chapter 5.

- `IsIsomorphicGraph( G, H )`
  Returns `true` when `G` is isomorphic to `H` and `false` otherwise. (B.9.16)
- `IsoMorphism( G, H )`
  Returns one isomorphism from `G` to `H`; `fail` if there is none. (B.9.20)
- `IsoMorphisms( G, H )`
  Returns the list of all isomorphisms from `G` to `H`. (B.9.21)
- `NextIsoMorphism( G, H, F )`
  Returns the next isomorphism (after `F`) from `G` to `H`. (B.14.1)
- `NextPropertyMorphism( G, H, F, PropList )`
  Returns the next morphism (after `F`) from `G` to `H` satisfying the list of properties `PropList`. (B.14.2)
- `PropertyMorphism( G, H, PropList )`
  Returns the first morphism from `G` to `H` satisfying the list of properties `PropList`. (B.16.9)

- `PropertyMorphisms( G, H, PropList )`
  Returns all morphisms from `G` to `H` satisfying the list of properties `PropList`. (B.16.10)

A.11 Graph Categories

- `CopyGraph( G )`
  Returns a fresh copy of `G`. Useful to change the category of a graph. (B.3.20)
- `GraphCategory( [ G, ... ] );`
  For internal use. Returns the minimal common category to a list of graphs. (B.7.12)
- `Graphs()`
  The category of all graphs that can be represented in YAGS. (B.7.13)
- `\in( G, Catgy )`
  `G in Catgy`
  Returns `true` if graph `G` belongs to category `Catgy` and `false` otherwise. (B.9.3)
- `LooplessGraphs()`
  The category of all graph that may contain arrows and edges but no loops. (B.12.4)
- `OrientedGraphs()`
  The category of all graphs that may contain arrows but no edges or loops. (B.15.5)
- `SetDefaultGraphCategory( Catgy )`
  Sets the default graph category to `Catgy`. (B.19.2)
- `SimpleGraphs()`
  The category of all graphs which may contain edges but no arrows or loops. (B.19.3)
- `TargetGraphCategory( [ G, ... ] );`
  For internal use. Within YAGS methods, returns the graph category to which the new graph will belong. (B.20.1)
- `UndirectedGraphs()`
  The category of all graphs that may contain loops and edges but no arrows. (B.21.3)

A.12 Digraphs

- `InNeigh( G, x )`
  Returns the list of in-neighbors of `x` in `G`. (B.9.5)
- `IsTournament( G )`
  Returns `true` if `G` contains no loops or edges but only arrows and it is maximal w.r.t. this property. (B.9.25)
- `IsTransitiveTournament( G )`
  Returns `true` if `G` is a Tournament and whenever xy and yz are arrows, then xz is an arrow too. (B.9.26)

- `Orientations( G )`
  Returns the list of all the oriented graphs that are obtained from `G` by replacing each edge by one arrow. (B.15.4)

- `OutNeigh( G, x )`
  Returns the list of out-neighbors of `x` in `G`. (B.15.6)

- `PaleyTournament( prime )`
  Returns the Paley tournament associated with prime number `prime`. (B.16.1)

A.13 Groups and Rings

- `CayleyGraph( Grp )`
  `CayleyGraph( Grp, Elms )`
  Returns the CayleyGraph of group `Grp`. (B.3.1)

- `Circulant( n, Jumps )`
  Returns the minimal graph invariant under the cyclic permutation $(1\ 2\ \cdots\ n)$ such that the vertex 1 is adjacent to the vertices in `Jumps`. (B.3.3)

- `GroupGraph( G, Grp )`
  `GroupGraph( G, Grp, Act )`
  Returns the minimal `Grp`-invariant (under the action `Act`) graph containing `G`. (B.7.18)

- `QuadraticRingGraph( Rng )`
  Returns a graph `G` whose vertices are the elements of the ring `Rng` and $xy \in E(G) \iff x+z^2 = y$ for some z in `Rng`. (B.17.2)

- `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`. (B.18.10)

- `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`. (B.21.4)

A.14 Backtracking

- `Backtrack( L, Opts, Chk, Done, Extra )`
  Returns the next solution (after `L`) to a backtracking combinatorial problem specified by its parameters. (B.2.1)

- `BacktrackBag( Opts, Chk, Done, Extra )`
  Returns the list of all solutions to a backtracking combinatorial problem specified by its parameters. (B.2.2)

A.15 Miscellaneous

- `DumpObject( Obj )`
  For internal use. Dumps all information available for object *Obj*. (B.4.16)

- `EasyExec( Dir, ProgName, InString )`
  `EasyExec( ProgName, InString )`
  Calls the external program *ProgName* with input string *InString*; returns the output string. (B.5.1)

- `EquivalenceRepresentatives( L, Eqiv )`
  Returns a sublist of *L*, which is a complete list of representatives of *L* under the equivalent relation *Equiv*. (B.5.4)

- `IsBoolean( Obj )`
  Returns `true` if object *Obj* is `true` or `false` and `false` otherwise. (B.9.8)

- `QtfyIsSimple( G )`
  For internal use. Returns how far is graph *G* from being simple. (B.17.1)

- `RandomlyPermuted( Obj )`
  Returns a copy of *Obj* with the order of its elements permuted randomly. (B.18.6)

- `RandomPermutation( n )`
  Returns a random permutation of the list `[1,2, ..., n]`. (B.18.4)

- `RandomSubset( Set )`
  `RandomSubset( Set, k )`
  `RandomSubset( Set, p )`
  Returns a random subset of the set *Set*. It also works for lists though. (B.18.5)

- `TimeInSeconds()`
  Returns the time in seconds since 1970-01-01 00:00:00 UTC as an integer. (B.20.3)

- `UFFind( UFS, x )`
  For internal use. Implements the *find* operation on the *union-find structure*. (B.21.1)

- `UFUnite( UFS, x, y )`
  For internal use. Implements the *unite* operation on the *union-find structure*. (B.21.2)

- `YAGSExec( ProgName, InString )`
  For internal use. Calls external program *ProgName* located in directory `YAGS-DIR/bin/` feeding it with *InString* as input and returning the output of the external program as a string. (B.24.1)

- `YAGSInfo`
  Global record where much YAGS-related information is stored. (B.24.2)

- `YAGSInfo.InfoClass`
  YAGS's progress reporting `InfoClass`. Several algorithms in YAGS report progress at `InfoLevel` 1 or 3. (B.24.3)

- `YAGSInfo.InfoOutput`
  Output file (or device) for YAGS's progress reporting `InfoClass`. (B.24.4)

A.16 Deprecated

We declare in this section the operations that, with higher probability, may disappear or change in a non-compatible manner in the future.

- `AutGroupGraph( G )`
  Returns the automorphism group of graph `G`. Use `AutomorphismGroup( G )` instead. (B.1.8)

- `DeclareQtfyProperty( Name, Filter )`
  For internal use. Declare a quantifiable property. (B.4.2)

- `DumpObject( Obj )`
  For internal use. Dumps all information available for object `Obj`. (B.4.16)

- `EasyExec( Dir, ProgName, InString )`
  `EasyExec( ProgName, InString )`
  Calls the external program `ProgName` with input string `InString`; returns the output string. (B.5.1)

- `GraphToRaw( FileName, G )`
  Writes the graph `G` in raw format to the file `FileName`. (B.7.16)

- `GraphUpdateFromRaw( FileName, G )`
  Updates the coordinates of `G` from a file `FileName` in raw format. (B.7.17)

- `GroupGraph( G, Grp )`
  `GroupGraph( G, Grp, Act )`
  Returns the minimal `Grp`-invariant (under the action `Act`) graph containing `G`. (B.7.18)

- `QtfyIsSimple( G )`
  For internal use. Returns how far is graph `G` from being simple. (B.17.1)

- `TimeInSeconds()`
  Returns the time in seconds since 1970-01-01 00:00:00 UTC as an integer. (B.20.3)

- `YAGSExec( ProgName, InString )`
  For internal use. Calls external program `ProgName` located in directory `YAGS-DIR/bin/` feeding it with `InString` as input and returning the output of the external program as a string. (B.24.1)

- `YAGSInfo`
  Global record where much YAGS-related information is stored. (B.24.2)

Appendix B

YAGS Functions Reference

This chapter contains a list of most YAGS's functions, with full definitions, in alphabetical order; but the predefined types of morphisms are best described in their own Section 5.2.

B.1 A

B.1.1 AddEdges

▷ `AddEdges(G, E)` (operation)

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

Example

```
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 ] ] )
```

B.1.2 AddVerticesByAdjacencies

▷ `AddVerticesByAdjacencies(G, NewAdjList)` (operation)

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.

Example

```
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 ] ] )
```

B.1.3 Adjacencies

▷ `Adjacencies(G)` (operation)

Returns the adjacency lists of graph `G`.

Example

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

B.1.4 Adjacency

▷ `Adjacency(G, x)` (operation)

Returns the adjacency list of vertex `x` in `G`.

Example

```
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 ]
```

B.1.5 AdjMatrix

▷ `AdjMatrix(G)` (attribute)

Returns the adjacency matrix of the graph `G`.

Example

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

B.1.6 AGraph

▷ `AGraph` (global variable)

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

Example

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

B.1.7 AntennaGraph

▷ AntennaGraph (global variable)

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

Example

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

B.1.8 AutomorphismGroup

▷ AutomorphismGroup(*G*) (attribute)
▷ AutGroupGraph(*G*) (attribute)

Returns the group of automorphisms of the graph *G*. Both forms are synonyms.

Example

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

B.2 B

B.2.1 Backtrack

▷ Backtrack(*L*, *Opts*, *Chk*, *Done*, *Extra*) (operation)

Generic, user-customizable backtracking algorithm.

The non-expert programmer is advised to read Chapter 6 first.

A backtracking 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 (children) 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 vice versa), 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 derangements (permutations of a set, where none of the elements appears in its original position).

Example

```
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
```

This operation reports progress at InfoLevel 3 (see B.24.3 and Section 6.4).

Extensive information on Backtrack and BacktrackBag can be found in Chapter 6.

B.2.2 BacktrackBag

▷ BacktrackBag(*Opts, Chk, Done, Extra*) (operation)

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

The following example computes all derangements of order 4.

Example

```
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 ] ]
```

This operation reports progress at `InfoLevel` 3 (see B.24.3 and Section 6.4).
Extensive information on `Backtrack` and `BacktrackBag` can be found in Chapter 6.

B.2.3 Basement

▷ `Basement(G, KnG, x)` (operation)
▷ `Basement(G, KnG, V)` (operation)

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* [23]. Loosely speaking, the basement of *x* is the set of vertices of *G* that constitutes the iterated clique *x*.

Example

```
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 ]
```

Formally, taking `m=n-1`, the basement is defined as follows:

```
Basement(G,G,x):=x;
Basement(G,KG,x):=VertexNames(KG)[x];
Basement(G,KnG,x):= Union(List(VertexNames(KnG)[x]), z->Basement(G,KmG,z));
```

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*.

Example

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

Basements have been used to study distances and diameters of clique graphs in [3] and [23].

B.2.4 BoundaryVertices

▷ `BoundaryVertices(G)` (attribute)

When *G* is (an underlying graph of a Whitney triangulation of) a compact surface, it returns the list of vertices in the boundary (of the triangulation) of the surface. That is, the list of vertices of *G* whose link is isomorphic to a path of length at least 2. It returns `fail` if *G* is not a compact surface.

Example

```
gap> BoundaryVertices(WheelGraph(4,2));
[ 6, 7, 8, 9 ]
gap> BoundaryVertices(Octahedron);
[  ]
```

B.2.5 BoxProduct

▷ BoxProduct(*G*, *H*) (operation)

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 $y = y'$.

Example

```
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 ] ]
```

B.2.6 BoxTimesProduct

▷ BoxTimesProduct(*G*, *H*) (operation)

Returns the boxtimes product, $G \boxtimes H$, of two graphs *G* and *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 $y \simeq y'$.

Example

```
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 ] ]
```

B.2.7 BullGraph

▷ BullGraph (global variable)

A triangle with two pendant vertices (horns).

Example

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

B.3 C

B.3.1 CayleyGraph

▷ CayleyGraph(*Grp*, *Elms*) (operation)
▷ CayleyGraph(*Grp*) (operation)

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.

Example

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

B.3.2 ChairGraph

▷ ChairGraph (global variable)

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

Example

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

B.3.3 Circulant

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

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 ist of *Jumps*.

Example

```
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 ] ] )
```

B.3.4 ClawGraph

▷ `ClawGraph` (global variable)

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

Example

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

B.3.5 CliqueGraph

▷ `CliqueGraph(`*G*`)` (attribute)
▷ `CliqueGraph(`*G*`, `*maxNumCli*`)` (operation)

Returns the intersection graph, $K(G)$, 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. However, the clique graph of *G* is returned if it has been computed earlier, regardless of the value of *maxNumCli*.

Example

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

This operation reports progress at InfoLevel 1 (see B.24.3).

B.3.6 CliqueNumber

▷ CliqueNumber(*G*) (attribute)

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

Example

```
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, 3, 4 ], [ 1, 2, 8 ], [ 4, 5, 6 ], [ 6, 7, 8 ] ]
gap> CliqueNumber(g);
4
```

This operation reports progress at InfoLevel 1 (see B.24.3).

B.3.7 Cliques

▷ Cliques(*G*) (attribute)
▷ Cliques(*G*, *maxNumCli*) (operation)

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 [4].

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

Example

```
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 ] ]
```

This operation reports progress at InfoLevel 1 (see B.24.3).

B.3.8 ClockworkGraph

▷ ClockworkGraph(*NNFSList*) (operation)
▷ ClockworkGraph(*NNFSList*, *rank*) (operation)
▷ ClockworkGraph(*NNFSList*, *Perm*) (operation)
▷ ClockworkGraph(*NNFSList*, *rank*, *Perm*) (operation)

Returns the clockwork graph [14][16] specified by its parameters.

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 [6], clique-divergent graphs of linear and polynomial growth rate [12][14],

clique-convergent graphs whose period is not invariant under removal of dominated vertices [7], clique-convergent graphs which become clique-divergent by just gluing a 4-cycle to a vertex [8], rank-divergent graphs [17], etc.

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:

Example

```
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 segments `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 exactly 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:

Example

```
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.

B.3.9 ComplementGraph

▷ `ComplementGraph(G)` (attribute)

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

Example

```
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 ] ] )
```

B.3.10 CompleteBipartiteGraph

▷ `CompleteBipartiteGraph(n, m)` (function)

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*.

Example

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

B.3.11 CompleteGraph

▷ `CompleteGraph(n)` (function)

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

Example

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

B.3.12 CompletelyParedGraph

▷ `CompletelyParedGraph(G)` (operation)

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

Example

```
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 := [ [  ] ] )
```

This operation reports progress at `InfoLevel` 1 (see B.24.3).

B.3.13 CompleteMultipartiteGraph

▷ `CompleteMultipartiteGraph(n1, n2[, n3, ...])` (function)

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*, ...

Example

```
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 ] ] )
```

B.3.14 CompletesOfGivenOrder

▷ `CompletesOfGivenOrder(G, ord)` (operation)

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

Example

```
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 ] ]
```

This operation reports progress at `InfoLevel` 3 (see B.24.3 and Section 6.4).

B.3.15 Composition

▷ `Composition(G, H)` (operation)

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

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

Example

```
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 ] ] )
```

B.3.16 Cone

▷ `Cone(G)` (operation)

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.

Example

```
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 ] ] )
```

B.3.17 ConnectedComponents

▷ `ConnectedComponents(G)` (attribute)

Returns the *connected components* of `G`.

Two vertices in a graph are *reachable* (from each other) if there is a path connecting them. Two vertices are in the same connected component iff they are reachable from each other. This operation thus computes the equivalence partition of the equivalence relation "reachable".

Example

```
gap> g:=GraphByWalks([3,1,4],[5,2]);
Graph( Category := SimpleGraphs, Order := 5, Size :=
3, Adjacencies := [ [ 3, 4 ], [ 5 ], [ 1 ], [ 1 ], [ 2 ] ] )
gap> ConnectedComponents(g);
[ [ 1, 3, 4 ], [ 2, 5 ] ]
gap> g1:=Composition(DiscreteGraph(3),g);
Graph( Category := SimpleGraphs, Order := 15, Size :=
9, Adjacencies := [ [ 3, 4 ], [ 5 ], [ 1 ], [ 1 ], [ 2 ], [ 8, 9 ],
  [ 10 ], [ 6 ], [ 6 ], [ 7 ], [ 13, 14 ], [ 15 ], [ 11 ], [ 11 ],
  [ 12 ] ] )
gap> ConnectedComponents(g1);
[ [ 1, 3, 4 ], [ 2, 5 ], [ 6, 8, 9 ], [ 7, 10 ], [ 11, 13, 14 ],
  [ 12, 15 ] ]
```

B.3.18 ConnectedGraphsOfGivenOrder

▷ `ConnectedGraphsOfGivenOrder(n)` (operation)

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

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

The 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 up to 9.

Example

```
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
```

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-DIR/data/`.

Example

```
gap> ConnectedGraphsOfGivenOrder(10);
#W Unreadable File: /opt/gap4r8/pkg/yags/data/graph10c.g6
fail
```

B.3.19 Coordinates

▷ `Coordinates(`*G*`)` (operation)

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*.

Example

```
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 ] ]
```

B.3.20 CopyGraph

▷ `CopyGraph(`*G*`)` (operation)

Returns a fresh copy of the 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`.

Example

```
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 ] ] )
```

B.3.21 Cube

▷ `Cube` (global variable)

The 1-skeleton of Plato's cube.

Example

```
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 ] ] )
```

B.3.22 CubeGraph

▷ `CubeGraph(n)` (function)

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

Example

```
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 ] ] )
```

B.3.23 CycleGraph

▷ `CycleGraph(n)` (function)

Returns the cyclic graph on n vertices.

Example

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

B.3.24 CylinderGraph

▷ `CylinderGraph(b, h)` (function)

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 boundary vertices are of degree 4.

Example

```
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 ] ] )
```

B.4 D

B.4.1 DartGraph

▷ DartGraph (global variable)

A diamond with a pendant vertex and maximum degree 4.

Example

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

B.4.2 DeclareQtfyProperty

▷ DeclareQtfyProperty(*Name*, *Filter*) (function)

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.

Example

```
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
```

B.4.3 Diameter

▷ `Diameter(G)` (attribute)

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

Example

```
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
```

B.4.4 DiamondGraph

▷ `DiamondGraph` (global variable)

The graph on 4 vertices and 5 edges.

Example

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

B.4.5 DiscreteGraph

▷ `DiscreteGraph(n)` (function)

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

Example

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

B.4.6 DisjointUnion

▷ `DisjointUnion(G, H)` (operation)

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

Example

```
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 ] ] )
```

B.4.7 Distance

▷ Distance(*G*, *x*, *y*) (operation)

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

Example

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

B.4.8 Distances

▷ Distances(*G*, *A*, *B*) (operation)

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.

Example

```
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 ]
```

B.4.9 DistanceGraph

▷ DistanceGraph(*G*, *Dist*) (operation)

Given a graph *G* and a 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*.

Example

```
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 ] ] )
```

B.4.10 DistanceMatrix

▷ `DistanceMatrix(G)` (attribute)

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.

Example

```
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 ] ]
```

B.4.11 DistanceSet

▷ `DistanceSet(G, A, B)` (operation)

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`.

Example

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

B.4.12 Dodecahedron

▷ `Dodecahedron` (global variable)

The 1-skeleton of Plato's Dodecahedron.

Example

```
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 ] ] )
```

B.4.13 DominatedVertices

▷ DominatedVertices(*G*) (attribute)

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.

Example

```
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 ]
```

B.4.14 DominoGraph

▷ DominoGraph (global variable)

Two squares glued by an edge.

Example

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

B.4.15 Draw

▷ Draw(*G*) (operation)

Takes a graph *G* and makes a drawing of it in a separate window. The user can then view and modify the drawing and finally 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 keyboard 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).

Example

```
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 ] ]
```

`Draw()` uses an external Java program (included with YAGS) and hence, may not work on some platforms.

Current version has been tested successfully on GNU/Linux, Mac OS X and Windows7. For other platforms (specially 32-bit platforms), you should probably (at least) set up correctly the variables `YAGSInfo.Draw.prog` and `YAGSInfo.Draw.opts`. The former is a string representing the external binary program path and name; the latter is a list of strings representing the required command line options. Java binaries are provided for 32 and 64 bit versions of GNU/Linux (which also works for Mac OS X) and of MS Windows.

Example

```
gap> YAGSInfo.Draw.prog; YAGSInfo.Draw.opts;
"/opt/gap4r8/pkg/yags/bin/draw/application.linux64/draw"
[  ]
```

The source code for the external program, made using *processing* (http://processing.org), is `YAGS-DIR/bin/draw/draw.pde`

B.4.16 DumpObject

▷ `DumpObject(`*`Obj`*`)` (operation)

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.

Example

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

B.5 E

B.5.1 EasyExec

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

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 directory objects. If *Dir* is not provided, *ProgName* must be in the system's binary PATH. If the program could not be located, `fail` is returned.

Example

```
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"
```

This operation have not been tested on MS Windows.

B.5.2 Eccentricity

▷ `Eccentricity(G, x)` (function)

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

Example

```
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
```

B.5.3 Edges

▷ `Edges(G)` (operation)

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

Example

```
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*.

Example

```
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 ] ]
```

B.5.4 EquivalenceRepresentatives

▷ EquivalenceRepresentatives(*L*, *Eqiv*) (operation)

Returns a sublist of *L*, which is a complete list of representatives of *L* under the equivalent relation *Equiv*.

Example

```
gap> L:=[10,2,6,5,9,7,3,1,4,8];
[ 10, 2, 6, 5, 9, 7, 3, 1, 4, 8 ]
gap> EquivalenceRepresentatives(L,function(x,y) return (x mod 4)=(y mod 4); end);
[ 10, 5, 7, 4 ]
gap> L:=Links(SnubDisphenoid);;Length(L);
8
gap> L:=EquivalenceRepresentatives(L,IsIsomorphicGraph);;Length(L);
2
gap> L;
[ 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 ] ] ) ]
```

B.6 F

B.6.1 FanGraph

▷ FanGraph(*n*) (function)

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

Example

```
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 ] ] )
```

B.6.2 FishGraph

▷ FishGraph (global variable)

A square and a triangle glued by a vertex.

Example

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

B.7 G

B.7.1 GemGraph

▷ GemGraph (global variable)

The 3-fan graph.

Example

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

B.7.2 Girth

▷ Girth(*G*) (attribute)

Returns the length of a minimum cycle in *G*. At this time, Girth is defined only for SimpleGraphs (B.19.3) and UndirectedGraphs (B.21.3). If *G* has loops, its girth is 1 by definition.

Example

```
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
```

B.7.3 Graph

▷ Graph(*Rec*) (operation)

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*.

Example

```
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.

Example

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

B.7.4 GraphAttributeStatistics

▷ GraphAttributeStatistics(*OrderList, ProbList, Attribute*) (function)

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:

Example

```
gap> GraphAttributeStatistics(10,1/2,IsCliqueHelly);
32
```

This tells us that 32 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:

Example

```
gap> GraphAttributeStatistics(10,1/10*[1..9],IsCliqueHelly);
[ 100, 100, 94, 63, 34, 16, 30, 76, 95 ]
```

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 34 in the previous example does not match the 32 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:

Example

```
gap> GraphAttributeStatistics([10,12..20],1/10*[1..9],IsCliqueHelly);
[ [ 100, 100, 92, 62, 37, 16, 36, 70, 97 ],
  [ 100, 99, 83, 34, 8, 1, 19, 68, 97 ],
```

```
    [ 100, 96, 54, 4, 2, 0, 6, 54, 98 ],
    [ 100, 89, 26, 2, 0, 0, 9, 42, 96 ],
    [ 100, 70, 13, 1, 0, 0, 6, 24, 94 ],
    [ 99, 70, 5, 0, 0, 0, 4, 22, 92 ] ]
gap> Display(last);
[ [  100,  100,   92,   62,   37,   16,   36,   70,   97 ],
    [  100,   99,   83,   34,    8,    1,   19,   68,   97 ],
    [  100,   96,   54,    4,    2,    0,    6,   54,   98 ],
    [  100,   89,   26,    2,    0,    0,    9,   42,   96 ],
    [  100,   70,   13,    1,    0,    0,    6,   24,   94 ],
    [   99,   70,    5,    0,    0,    0,    4,   22,   92 ] ]
```

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:

Example

```
gap> GraphAttributeStatistics(10,1/2,Diameter);
[ [ 2, 34 ], [ 3, 59 ], [ 4, 5 ], [ 5, 1 ], [ infinity, 1 ] ]
```

The returned statistics mean that among the 100 generated random graphs on 10 vertices with edge probability 1/2, there were 34 graphs with diameter 2, 59 graphs of diameter 3, 5 of 4, 1 of 5 and there was one graph which was 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`*:

Example

```
gap> GraphAttributeStatistics(10,1/5*[1..4],Diameter);
[ [ [ 3, 1 ], [ 4, 7 ], [ 5, 8 ], [ 6, 6 ], [ infinity, 78 ] ],
  [ [ 2, 6 ], [ 3, 55 ], [ 4, 21 ], [ 5, 1 ], [ 6, 1 ],
      [ infinity, 16 ] ], [ [ 2, 74 ], [ 3, 25 ], [ 4, 1 ] ],
  [ [ 2, 100 ] ] ]
gap> GraphAttributeStatistics([10,12,14],1/5*[1..4],Diameter);
[ [ [ [ 3, 2 ], [ 4, 8 ], [ 5, 11 ], [ 6, 5 ], [ 7, 1 ],
          [ infinity, 73 ] ],
      [ [ 2, 6 ], [ 3, 56 ], [ 4, 23 ], [ 5, 7 ], [ infinity, 8 ] ],
      [ [ 2, 72 ], [ 3, 27 ], [ infinity, 1 ] ],
      [ [ 2, 99 ], [ 3, 1 ] ] ],
  [
      [ [ 3, 4 ], [ 4, 13 ], [ 5, 10 ], [ 6, 6 ], [ 7, 3 ],
          [ infinity, 64 ] ],
      [ [ 2, 7 ], [ 3, 69 ], [ 4, 17 ], [ infinity, 7 ] ],
      [ [ 2, 76 ], [ 3, 24 ] ], [ [ 2, 100 ] ] ],
  [ [ [ 4, 12 ], [ 5, 16 ], [ 6, 7 ], [ 7, 3 ], [ infinity, 62 ] ],
      [ [ 2, 8 ], [ 3, 86 ], [ 4, 4 ], [ infinity, 2 ] ],
      [ [ 2, 86 ], [ 3, 14 ] ], [ [ 2, 100 ] ] ] ]
```

B.7.5 Graph6ToGraph

▷ `Graph6ToGraph(`*`String`*`)` (operation)

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.

Example

```
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 (B.9.2).

B.7.6 GraphByAdjacencies

▷ GraphByAdjacencies(*AdjList*) (function)

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

Example

```
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.

Example

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

B.7.7 GraphByAdjMatrix

▷ GraphByAdjMatrix(*Mat*) (function)

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

Example

```
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`.

Example

```
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 ] ]
```

B.7.8 GraphByCompleteCover

▷ `GraphByCompleteCover(`*Cover*`)` (function)

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

Example

```
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 ] ] )
```

B.7.9 GraphByEdges

▷ `GraphByEdges(`*L*`)` (function)

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

Example

```
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*.

Example

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

Note that `GraphByWalks` (B.7.11) can do the same and much more.

B.7.10 GraphByRelation

▷ GraphByRelation(*V, Rel*) (function)
▷ GraphByRelation(*n, Rel*) (function)

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 $\{1,2,\ldots,n\}$.

Example

```
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 ] ] )
```

B.7.11 GraphByWalks

▷ GraphByWalks(*Walk1, Walk2, ...*) (function)

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

Example

```
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.

Example

```
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.

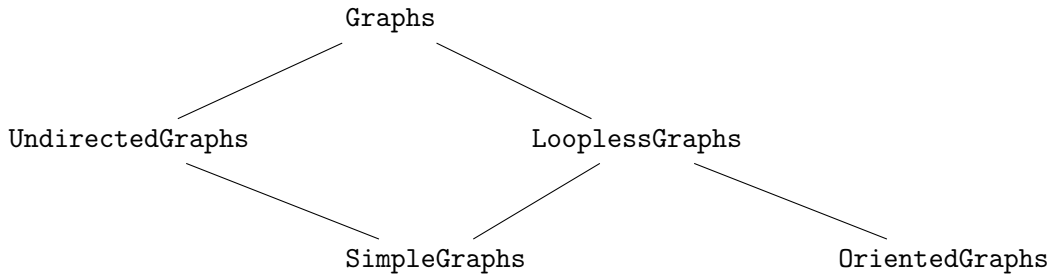
Example

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

B.7.12 GraphCategory

▷ GraphCategory(*[G, ...]*) (function)

For internal use. Returns the minimal common graph 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:



Example

```
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]);
<Category "Graphs">
gap> GraphCategory([g1,g2]);
<Category "LooplessGraphs">
gap> GraphCategory([g1,g3]);
<Category "UndirectedGraphs">
```

B.7.13 Graphs

▷ `Graphs(`*G*`)` (function)

`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.

Example

```
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 ] ] )
```

B.7.14 GraphsOfGivenOrder

▷ `GraphsOfGivenOrder(`*n*`)` (operation)

Returns the list of all graphs of order *n* (up to 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 up to 9.

Example

```
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
```

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-DIR/data/.

Example

```
gap> GraphsOfGivenOrder(10);
#W Unreadable File: /opt/gap4r8/pkg/yags/data/graph10.g6
fail
```

B.7.15 GraphSum

▷ GraphSum(*G*, *L*) (operation)

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

The lexicographic sum is computed as follows:

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

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

Example

```
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 ] ] )
```

B.7.16 GraphToRaw

▷ `GraphToRaw(`*`FileName, G`*`)` (operation)

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` (B.4.15)). Intended for internal use only.

Example

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

B.7.17 GraphUpdateFromRaw

▷ `GraphUpdateFromRaw(`*`FileName, G`*`)` (operation)

Updates the coordinates of *G* from a file *FileName* in raw format as written by `draw` (see `Draw` (B.4.15)). Intended for internal use only.

B.7.18 GroupGraph

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

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

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

Example

```
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 ]
 ] )
```

B.8 H

B.8.1 HararyToMcKay

▷ `HararyToMcKay(`*`Spec`*`)` (operation)
▷ `McKayToHarary(`*`index`*`)` (operation)

Returns the McKay's *index* of a Harary's graph specification *Spec* and vice versa. Frank Harary published in his book [10], a list of all 208 simple graphs of order up to 6 (up to 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 up to 10 (also up to 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 vice versa.

Example

```
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
```

Example

```
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 ] ]
gap> McKayToHarary(209);
fail
```

B.8.2 HouseGraph

▷ HouseGraph (global variable)

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

Example

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

B.9 I

B.9.1 Icosahedron

▷ `Icosahedron` (global variable)

The 1-skeleton of Plato's icosahedron.

Example

```
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 ] ] )
```

B.9.2 ImportGraph6

▷ `ImportGraph6(`*`Filename`*`)` (operation)

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. It is also assumed that your Operative System is a Unix-like system.

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 ] ] ) ]
```

See also `Graph6ToGraph` (B.7.5).

B.9.3 in

▷ `in(`*`G, Catgy`*`)` (operation)

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

Example

```
gap> g:=WheelGraph(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 ] ] )
gap> g in SimpleGraphs;
true
gap> g in Graphs;
true
gap> g in OrientedGraphs;
false
```

B.9.4 InducedSubgraph

▷ `InducedSubgraph(G, V)` (operation)

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

Example

```
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.

Example

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

B.9.5 InNeigh

▷ `InNeigh(G, x)` (operation)

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

Example

```
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 ]
```

B.9.6 InteriorVertices

▷ `InteriorVertices(G)` (attribute)

When `G` is (an underlying graph of a Whitney triangulation of) a compact surface, it returns the list of vertices in the interior (of the triangulation) of the surface. That is, the list of vertices of `G` that have links isomorphic to a cycle. It returns `fail` if `G` is not a compact surface.

Example

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

B.9.7 IntersectionGraph

▷ `IntersectionGraph(L)` (function)

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.

Example

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

B.9.8 IsBoolean

▷ `IsBoolean(Obj)` (function)

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

Example

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

B.9.9 IsCliqueGated

▷ `IsCliqueGated(G)` (property)

Returns `true` if `G` is a clique gated graph [9].
This operation reports progress at `InfoLevel` 1 (see B.24.3).

B.9.10 IsCliqueHelly

▷ `IsCliqueHelly(G)` (property)

Returns `true` if the set of (maximal) cliques `G` satisfy the *Helly* property.
The Helly property is defined as follows:

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

Here we use the Dragan-Szwarcfiter characterization [5][29] to compute the Helly property.

Example

```
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
```

B.9.11 IsCompactSurface

▷ `IsCompactSurface(G)` (property)

Returns `true` if every link of `G` is either an `n`-cycle, for $n \geq 4$ or an `m`-path, for $m \geq 2$. (not necessarily the same `n`/`m` for all vertices); it returns `false` otherwise.

This notion correspond to Whitney triangulations of compact surfaces [15] in which the (maximal) cliques of the graph are exactly the triangles of the triangulation.

Example

```
gap> IsCompactSurface(Icosahedron);
true
gap> IsCompactSurface(RemoveVertices(Icosahedron,[1]));
true
gap> IsCompactSurface(WheelGraph(4,2));
true
gap> IsCompactSurface(Tetrahedron);
false
gap> IsCompactSurface(CompleteGraph(2));
false
gap> IsCompactSurface(CompleteGraph(3));
true
gap> IsCompactSurface(CompleteGraph(4));
false
```

Topologically, the difference between a surface and a compact surface is that the points of a surface always have a open neighborhood homeomorphic to an open disk, whereas a compact surface may also contain points with open neighborhoods homeomorphic to a closed half-plane.

B.9.12 IsComplete

▷ `IsComplete(G, L)` (operation)

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

Example

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

B.9.13 IsCompleteGraph

▷ `IsCompleteGraph(G)` (property)

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

B.9.14 IsDiamondFree

▷ `IsDiamondFree(G)` (property)

Returns `true` if `G` is free from induced diamonds (see `DiamondGraph` (B.4.4)); `false` otherwise.

Example

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

B.9.15 IsEdge

▷ `IsEdge(G, x, y)` (operation)
▷ `IsEdge(G, e)` (operation)

Returns `true` if `e:=[x,y]` is an edge of `G`.

Example

```
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 algorithms 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.

Example

```
gap> IsEdge(PathGraph(3),[7,3]);
false
gap> IsEdge(PathGraph(3),7,3);
Error, List Element: <list>[7] must have an assigned value
```

B.9.16 IsIsomorphicGraph

▷ `IsIsomorphicGraph(G, H)` (operation)

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

Example

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

B.9.17 IsLocallyConstant

▷ IsLocallyConstant(*G*) (property)

Returns true if all the links of *G* are isomorphic to each other; false otherwise.

Example

```
gap> IsLocallyConstant(PathGraph(2));
true
gap> IsLocallyConstant(PathGraph(3));
false
gap> IsLocallyConstant(CompleteGraph(3));
true
gap> IsLocallyConstant(CycleGraph(4));
true
gap> IsLocallyConstant(Icosahedron);
true
gap> IsLocallyConstant(TorusGraph(5,4));
true
gap> IsLocallyConstant(WheelGraph(4,2));
false
gap> IsLocallyConstant(SnubDisphenoid);
false
```

B.9.18 IsLocallyH

▷ IsLocallyH(*G*, *H*) (operation)

Returns true if all the links of *G* are isomorphic to *H*; false otherwise.

Example

```
gap> IsLocallyH(Octahedron,CycleGraph(4));
true
gap> IsLocallyH(Octahedron,CycleGraph(5));
false
gap> IsLocallyH(Icosahedron,CycleGraph(5));
true
gap> IsLocallyH(TorusGraph(4,4),CycleGraph(6));
true
```

B.9.19 IsLoopless

▷ IsLoopless(*G*) (property)

Returns true if the graph *G* have no loops; false otherwise. Loops are edges from a vertex to itself.

B.9.20 IsoMorphism

▷ `IsoMorphism(G, H)` (operation)

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)]`.

Example

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

See `NextIsoMorphism` (B.14.1).

B.9.21 IsoMorphisms

▷ `IsoMorphisms(G, H)` (operation)

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)]`.

Example

```
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 ] ]
```

B.9.22 IsOriented

▷ `IsOriented(G)` (property)

Returns `true` if the 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.

B.9.23 IsSimple

▷ `IsSimple(G)` (property)

Returns `true` if the 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.

B.9.24 IsSurface

▷ `IsSurface(G)` (property)

Returns `true` if every link of G is an n-cycle, for $n \geq 4$ (not necessarily the same n for all vertices); `false` otherwise.

This notion correspond to Whitney triangulations of (closed) surfaces [15] in which the (maximal) cliques of the graph are exactly the triangles of the triangulation.

Example

```
gap> IsSurface(SnubDisphenoid);
true
gap> IsSurface(Icosahedron);
true
gap> IsSurface(RemoveVertices(Icosahedron,[1]));
false
gap> IsSurface(TorusGraph(4,5));
true
gap> IsSurface(WheelGraph(4,2));
false
gap> IsSurface(Tetrahedron);
false
```

Topologically, the difference between a (closed) surface and a compact surface is that the points of a surface always have a open neighborhood homeomorphic to an open disk, whereas a compact surface may also contain points with open neighborhoods homeomorphic to a closed half-plane.

B.9.25 IsTournament

▷ `IsTournament(G)` (property)

Returns `true` if *G* is a tournament. A *tournament* is a graph without loops and such that for every pair of vertices `x`, `y`, either `[x,y]` is an arrow of *G* , or `[y,x]` is an arrow of *G*, but not both.

Example

```
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
```

B.9.26 IsTransitiveTournament

▷ `IsTransitiveTournament(G)` (property)

Returns `true` if *G* is a transitive tournament. A tournament is a *transitive tournament* if whenever `[x,y]` and `[y,z]` are arrows of the tournament, `[x,z]` is also an arrow of the tournament.

Example

```
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
```

B.9.27 IsUndirected

▷ `IsUndirected(G)` (property)

Returns `true` if the 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 edge of `G`.

B.10 J

B.10.1 JohnsonGraph

▷ `JohnsonGraph(n, r)` (function)

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.

Example

```
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 ] ]
```

B.10.2 Join

▷ `Join(G, H)` (operation)

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`.

Example

```
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 ] ] )
```

B.11 K

B.11.1 KiteGraph

▷ `KiteGraph` (global variable)

A diamond (see `DiamondGraph` (B.4.4)) with a pendant vertex and maximum degree 3.

Example

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

B.12 L

B.12.1 LineGraph

▷ `LineGraph(G)` (operation)

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.

Example

```
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 ] ] )
```

B.12.2 Link

▷ `Link(G, x)` (operation)

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

Example

```
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 ] ] )
```

B.12.3 Links

▷ `Links(G)` (attribute)

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

Example

```
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 ] ] ) ]
```

B.12.4 LooplessGraphs

▷ `LooplessGraphs(G)` (function)

`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`.

Example

```
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 ] ] )
```

B.13 M

B.13.1 MaxDegree

▷ `MaxDegree(G)` (operation)

Returns the maximum degree of a vertex in the graph *G*.

Example

```
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
```

B.13.2 MinDegree

▷ `MinDegree(G)` (operation)

Returns the minimum degree of a vertex in the graph *G*.

Example

```
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
```

B.14 N

B.14.1 NextIsoMorphism

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

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)]`.

Example

```
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 ]
```

B.14.2 NextPropertyMorphism

▷ `NextPropertyMorphism(G, H, F, PropList)` (operation)

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 be 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 (but, for technical reasons, `F` stores the list `[fail]` instead).

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)]`.

Example

```
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
```

This operation reports progress at InfoLevel 3 (see B.24.3 and Section 6.4).
Extensive information about graph morphisms can be found in Chapter 5.

B.14.3 NumberOfCliques

▷ NumberOfCliques(*G*) (attribute)
▷ NumberOfCliques(*G*, *maxNumCli*) (operation)

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.

Example

```
gap> NumberOfCliques(Icosahedron,15);
15
gap> NumberOfCliques(Icosahedron);
20
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. This test may take several minutes to complete:

Example

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

This operation reports progress at InfoLevel 1 (see B.24.3).

B.14.4 NumberOfConnectedComponents

▷ NumberOfConnectedComponents(*G*) (attribute)

Returns the number of connected components of *G*. See ConnectedComponents (B.3.17).

B.15 O

B.15.1 OctahedralGraph

▷ OctahedralGraph(*n*) (function)

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 2*n* vertices.

Example

```
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 ] ] )
```

B.15.2 Octahedron

▷ `Octahedron` (global variable)

The 1-skeleton of Plato's octahedron.

Example

```
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 ] ] )
```

B.15.3 Order

▷ `Order(`*G*`)` (attribute)

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

Example

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

B.15.4 Orientations

▷ `Orientations(`*G*`)` (operation)

Returns the list of all the oriented graphs that are obtained from *G* by replacing (in every possible way) each edge `[x,y]` of *G* by one arrow: either `[x,y]` or `[y,x]`. In each of these orientations the loops are removed and existing arrows of *G* are left untouched.

Note that this operation will use time and memory which is exponential on the number of edges of *G*.

Example

```
gap> g:=GraphByWalks([1,1,2,3,1,3,2]:GraphCategory:=Graphs);
Graph( Category := Graphs, Order := 3, Size := 6, Adjacencies :=
[ [ 1, 2, 3 ], [ 3 ], [ 1, 2 ] ] )
gap> Orientations(g);
[ Graph( Category := OrientedGraphs, Order := 3, Size :=
    3, Adjacencies := [ [ 2 ], [  ], [ 1, 2 ] ] ),
  Graph( Category := OrientedGraphs, Order := 3, Size :=
    3, Adjacencies := [ [ 2 ], [ 3 ], [ 1 ] ] ),
  Graph( Category := OrientedGraphs, Order := 3, Size :=
    3, Adjacencies := [ [ 2, 3 ], [  ], [ 2 ] ] ),
  Graph( Category := OrientedGraphs, Order := 3, Size :=
    3, Adjacencies := [ [ 2, 3 ], [ 3 ], [  ] ] ) ]
gap> Length(Orientations(Octahedron));
4096
```

Note that `Orientations( G )` returns a list of graphs, each of them in the category `OrientedGraphs` regardless of the `TargetGraphCategory`.

This operation reports progress at `InfoLevel` 3 (see B.24.3 and Section 6.4).

B.15.5 OrientedGraphs

▷ `OrientedGraphs(G)` (function)

`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`.

Example

```
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 ] ] )
```

B.15.6 OutNeigh

▷ `OutNeigh(G, x)` (operation)

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

Example

```
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 ]
```

B.16 P

B.16.1 PaleyTournament

▷ `PaleyTournament(prime)` (operation)

Returns the *Paley tournament* associated with prime number `prime`. The `prime` must be congruent to 3 mod 4. The Paley tournament is the oriented circulant whose `jumps` are all the squares of the ring $\mathbb{Z}_p$.

Example

```
gap> Filtered([1..30],x -> 0=((x-3) mod 4) and IsPrime(x));
[ 3, 7, 11, 19, 23 ]
gap> PaleyTournament(3);PaleyTournament(7);PaleyTournament(11);
Graph( Category := OrientedGraphs, Order := 3, Size :=
3, Adjacencies := [ [ 2 ], [ 3 ], [ 1 ] ] )
```

```
Graph( Category := OrientedGraphs, Order := 7, Size :=
21, Adjacencies := [ [ 2, 3, 5 ], [ 3, 4, 6 ], [ 4, 5, 7 ],
  [ 1, 5, 6 ], [ 2, 6, 7 ], [ 1, 3, 7 ], [ 1, 2, 4 ] ] )
Graph( Category := OrientedGraphs, Order := 11, Size :=
55, Adjacencies := [ [ 2, 4, 5, 6, 10 ], [ 3, 5, 6, 7, 11 ],
  [ 1, 4, 6, 7, 8 ], [ 2, 5, 7, 8, 9 ], [ 3, 6, 8, 9, 10 ],
  [ 4, 7, 9, 10, 11 ], [ 1, 5, 8, 10, 11 ], [ 1, 2, 6, 9, 11 ],
  [ 1, 2, 3, 7, 10 ], [ 2, 3, 4, 8, 11 ], [ 1, 3, 4, 5, 9 ] ] )
gap> PaleyTournament(5);
fail
```

Note that `PaleyTournament( prime )` returns a graph in the category `OrientedGraphs` regardless of the `TargetGraphCategory`.

B.16.2 ParachuteGraph

▷ `ParachuteGraph` (global variable)

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

Example

```
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 ] ] )
```

B.16.3 ParapluieGraph

▷ `ParapluieGraph` (global variable)

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

Example

```
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 ] ] )
```

B.16.4 ParedGraph

▷ `ParedGraph(G)` (operation)

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.

Example

```
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 := [ [  ] ] )
```

This operation reports progress at `InfoLevel` 1 (see B.24.3).

B.16.5 PathGraph

▷ `PathGraph(`*n*`)` (function)

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 ] ] )
```

B.16.6 PawGraph

▷ `PawGraph` (global variable)

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

Example

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

B.16.7 PetersenGraph

▷ `PetersenGraph` (global variable)

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

Example

```
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 ] ] )
```

B.16.8 PowerGraph

▷ `PowerGraph(`*G*`, `*exp*`)` (operation)

Returns the DistanceGraph (B.4.9) 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.

Example

```
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 ] ] )
```

B.16.9 PropertyMorphism

▷ PropertyMorphism(*G, H, PropList*) (operation)

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)].

Example

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

This operation reports progress at InfoLevel 3 (see B.24.3 and Section 6.4).

Extensive information about graph morphisms can be found in Chapter 5.

B.16.10 PropertyMorphisms

▷ PropertyMorphisms(*G, H, PropList*) (operation)

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)].

Example

```
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 ] ]
```

This operation reports progress at `InfoLevel` 3 (see B.24.3 and Section 6.4).
Extensive information about graph morphisms can be found in Chapter 5.

B.17 Q

B.17.1 QtfyIsSimple

▷ `QtfyIsSimple(`*G*`)` (attribute)

For internal use. Returns a non-negative integer indicating how far is the graph *G* from being a simple graph. The return value of 0 means the graph that the graph is simple.

B.17.2 QuadraticRingGraph

▷ `QuadraticRingGraph(`*Rng*`)` (operation)

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

Example

```
gap> QuadraticRingGraph(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 ] ] )
```

B.17.3 QuotientGraph

▷ `QuotientGraph(`*G, Part*`)` (operation)
▷ `QuotientGraph(`*G, L1, L2*`)` (operation)

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 omitted in *Part*.

Example

```
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.

Example

```
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 ] ] )
```

B.18 R

B.18.1 Radius

▷ Radius(*G*) (attribute)

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

Example

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

B.18.2 RandomCirculant

▷ RandomCirculant(*n*) (operation)
▷ RandomCirculant(*n*, *k*) (operation)
▷ RandomCirculant(*n*, *p*) (operation)

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. In the ambiguous case when the second parameter is 1, it is interpreted as the value of *k*.

Example

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

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

B.18.3 RandomGraph

▷ `RandomGraph(n, p)` (function)

▷ `RandomGraph(n)` (function)

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

Example

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

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

Example

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

B.18.4 RandomPermutation

▷ `RandomPermutation(n)` (operation)

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

Example

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

B.18.5 RandomSubset

▷ `RandomSubset(Set)` (operation)
▷ `RandomSubset(Set, k)` (operation)
▷ `RandomSubset(Set, p)` (operation)

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*.

Example

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

Example

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

B.18.6 RandomlyPermuted

▷ `RandomlyPermuted(Obj)` (operation)

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

Example

```
gap> RandomlyPermuted([1..9]);
[ 8, 7, 1, 9, 4, 2, 5, 6, 3 ]
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 := [ [ 2, 3 ], [ 1 ], [ 1, 4 ], [ 3 ] ] )
```

B.18.7 RemoveEdges

▷ RemoveEdges(*G*, *E*) (operation)

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

Example

```
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 ] ] )
```

B.18.8 RemoveVertices

▷ RemoveVertices(*G*, *V*) (operation)

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

Example

```
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 ] ] )
```

B.18.9 RGraph

▷ RGraph (global variable)

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

Example

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

B.18.10 RingGraph

▷ RingGraph(*Rng*, *Elms*) (operation)

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*.

Example

```
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 ] ] )
```

B.19 S

B.19.1 SetCoordinates

▷ SetCoordinates(*G*, *Coord*) (operation)

Sets the coordinates of the vertices of *G*, which are used to draw *G* by Draw (B.4.15).

Example

```
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 ] ]
```

B.19.2 SetDefaultGraphCategory

▷ SetDefaultGraphCategory(*Catgy*) (function)

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 TargetGraphCategory, so loops may be removed, and arrows may replaced by edges or vice versa, depending on the category that the new graph belongs to.

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

Example

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

B.19.3 SimpleGraphs

▷ `SimpleGraphs(G)` (function)

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

Example

```
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 ] ] )
```

B.19.4 Size

▷ `Size(G)` (attribute)

Returns the number of edges of the graph `G`. Note that the returned value depends not only on the structure of the graph, but also on the category to which it belongs.

Example

```
gap> g1:=CycleGraph(4);
Graph( Category := SimpleGraphs, Order := 4, Size :=
4, Adjacencies := [ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
gap> g2:=CopyGraph(g1:GraphCategory:=Graphs);
Graph( Category := Graphs, Order := 4, Size := 8, Adjacencies :=
[ [ 2, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ] ] )
gap> Size(g1);
```

```
4
gap> Size(g2);
8
```

B.19.5 SnubDisphenoid

▷ SnubDisphenoid (global variable)

The 1-skeleton of the 84th Johnson solid.

Example

```
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 ] ] )
```

B.19.6 SpanningForest

▷ SpanningForest(*G*) (operation)

Returns the a maximal spanning forest of *G*. Since the forest is maximal, it is composed of a spanning tree for each connected component of *G*. In particular, this operation actually returns a spanning tree whenever the graph is connected.

B.19.7 SpanningForestEdges

▷ SpanningForestEdges(*G*) (operation)

Returns the edges of a maximal spanning forest of *G*. Since the forest is maximal, it is composed of a spanning tree for each connected component of *G*. In particular, this operation actually returns the edges of a spanning tree whenever the graph is connected.

B.19.8 SpikyGraph

▷ SpikyGraph(*n*) (function)

The spiky graph is constructed as follows: Take a 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 $Vertices(K_n)$.

Example

```
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 ] ] )
```

B.19.9 SunGraph

▷ `SunGraph(n)` (function)

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.

Example

```
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 ] ] )
```

B.19.10 Suspension

▷ `Suspension(G)` (operation)

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.

Example

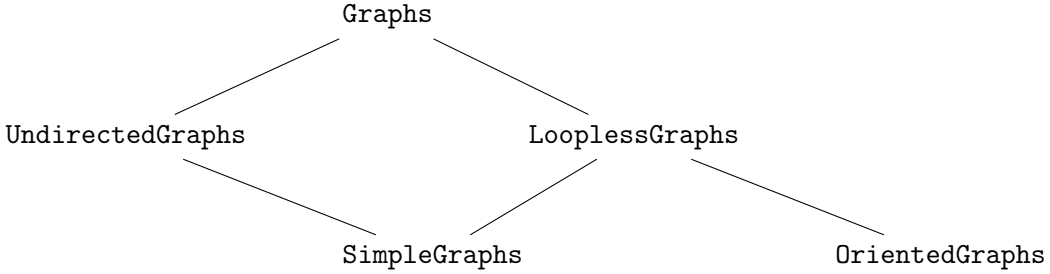
```
gap> Suspension(CycleGraph(4));
Graph( Category := SimpleGraphs, Order := 6, Size :=
12, Adjacencies := [ [ 3, 4, 5, 6 ], [ 3, 4, 5, 6 ], [ 1, 2, 4, 6 ],
  [ 1, 2, 3, 5 ], [ 1, 2, 4, 6 ], [ 1, 2, 3, 5 ] ] )
```

B.20 T

B.20.1 TargetGraphCategory

▷ `TargetGraphCategory([G, ...])` (function)

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:



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 vice versa, 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 TargetGraphCategory so that the user may indicate the graph category she/he wants for the new graph.

Example

```
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:

Example

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

B.20.2 Tetrahedron

▷ Tetrahedron (global variable)

The 1-skeleton of Plato's tetrahedron.

Example

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

B.20.3 TimeInSeconds

▷ TimeInSeconds() (operation)

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.

Example

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

Currently, this operation does not work on MS Windows.

B.20.4 TimesProduct

▷ `TimesProduct(G, H)` (operation)

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

Example

```
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 ] ]
```

B.20.5 TorusGraph

▷ `TorusGraph(n, m)` (function)

Returns (the underlying graph of) a triangulation of the torus on nm vertices. This graph 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 $n, m \geq 3$.

Example

```
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: the (maximal) cliques of the graph are exactly the triangles of the triangulation. The clique behavior of these graphs were extensively studied in [13]. However, this operation constructs the described graph for all $n, m \geq 1$.

Example

```
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.

B.20.6 TreeGraph

▷ `TreeGraph(`*arity, depth*`)` (operation)
▷ `TreeGraph(`*ArityList*`)` (operation)

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

Example

```
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 ] ] )
```

B.20.7 TrivialGraph

▷ `TrivialGraph` (global variable)

The one vertex graph.

Example

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

B.21 U

B.21.1 UFFind

▷ UFFind(*UFS*, *x*) (function)

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

B.21.2 UFUnite

▷ UFUnite(*UFS*, *x*, *y*) (function)

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

B.21.3 UndirectedGraphs

▷ UndirectedGraphs(*G*) (function)

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.

Example

```
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 ] ] )
```

B.21.4 UnitsRingGraph

▷ UnitsRingGraph(*Rng*) (operation)

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*.

Example

```
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 ] ] )
```

B.22 V

B.22.1 VertexDegree

▷ `VertexDegree(G, x)` (operation)

Returns the degree of vertex x in Graph `G`.

Example

```
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
```

B.22.2 VertexDegrees

▷ `VertexDegrees(G)` (operation)

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

Example

```
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 ]
```

B.22.3 VertexNames

▷ `VertexNames(G)` (attribute)

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

Example

```
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 ] ]
```

B.22.4 Vertices

▷ `Vertices(G)` (operation)

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

Example

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

B.23 W

B.23.1 WheelGraph

▷ `WheelGraph(n)` (operation)
▷ `WheelGraph(n, r)` (operation)

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.

Example

```
gap> WheelGraph(5);
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.

Example

```
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 ] ] )
```

B.24 Y

B.24.1 YAGSExec

▷ `YAGSExec(ProgName, InString)` (operation)

For internal use. Calls external program *ProgName* located in directory YAGS-DIR/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.

Example

```
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"
```

This operation have not been tested on MS Windows.

B.24.2 YAGSInfo

▷ YAGSInfo (global variable)

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.

Example

```
gap> YAGSInfo;
rec( Arch := 1, DataDirectory := "/opt/gap4r8/pkg/yags/data",
  Directory := "/opt/gap4r8/pkg/yags",
  Draw :=
    rec( opts := [  ],
      prog := "/opt/gap4r8/pkg/yags/bin/draw/application.linux64/draw" ),
  InfoClass := YAGSInfoClass, InfoOutput := "*stdout*", Version := "0.0.1",
  graph6 := rec( BinListToNum := function( L ) ... end,
      BinListToNumList := function( L ) ... end,
      HararyList := [ [ 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, 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 ] ], 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 ) )
```

B.24.3 YAGSInfo.InfoClass

▷ YAGSInfo.InfoClass (global variable)

YAGS uses the **Reference: InfoLevel** mechanism in some algorithms for progress reporting. This is useful in algorithms that may take a lot of time to finish, so the user is informed about how much work is already done and how much work remains to be done; this way, the user can decide whether to wait for the response or not.

Enabling and disabling progress reporting is done by changing the `InfoLevel` of `YAGSInfo.InfoClass` to the appropiate level. The default `InfoLevel` for `YAGSInfo.InfoClass` is 0, and some of YAGS algorithms report at `InfoLevel` 1, and others at `InfoLevel` 3.

Example

```
gap> SetInfoLevel(YAGSInfo.InfoClass,3);
gap> FullMonoMorphisms(PathGraph(3),CycleGraph(3));
#I [  ]
#I [ 1 ]
#I [ 1, 2 ]
#I [ 1, 3 ]
#I [ 2 ]
#I [ 2, 1 ]
#I [ 2, 3 ]
#I [ 3 ]
#I [ 3, 1 ]
#I [ 3, 2 ]
[  ]
gap> SetInfoLevel(YAGSInfo.InfoClass,0);
gap> FullMonoMorphisms(PathGraph(3),CycleGraph(3));
[  ]
```

The algorithms that report progress at `InfoLevel` 1 are `ParedGraph` (B.16.4) and `Cliques` (B.3.7), and also the algorithms that use those, namely: `CliqueGraph` (B.3.5), `CliqueNumber` (B.3.6), `CompletelyParedGraph` (B.3.12), `IsCliqueGated` (B.9.9) and `NumberOfCliques` (B.14.3).

The algorithms that report at `InfoLevel` 3 are `Backtrack` (B.2.1) and the algorithms that use that one, namely: `BacktrackBag` (B.2.2), `CompletesOfGivenOrder` (B.3.14), `Orientations` (B.15.4) and all the morphism-related operations in Chaper 5. The meaning of the progress strings reported in all these functions are described in Section 6.4.

The output of the progress info may be redirected to a file or character device by setting the variable `YAGSInfo.InfoOutput` (B.24.4) accordingly.

B.24.4 YAGSInfo.InfoOutput

▷ `YAGSInfo.InfoOutput` (global variable)

The output of the progress info reported by some algorithms (see `YAGSInfo.InfoClass` (B.24.3)) may be redirected to a file by setting the variable `YAGSInfo.InfoOutput` accordingly. The default value of `YAGSInfo.InfoOutput:="*stdout*"` means the console; but setting the name of a file as the value of `YAGSInfo.InfoOutput` sends the output to that file. In Unix-like systems, we can also use the name of a character device (like `"/dev/null"`, `"/dev/tty"` or `"/dev/pts/1"`) to redirect the progress info output to that device.

References

[1] L. Alcón, L. Faria, C. de Figueiredo and M. Gutierrez. *The complexity of clique graph recognition*. Theoret. Comput. Sci. **410** (2009) 2072 – 2083. 17

[2] B. Bollobás. *Random graphs*, volume 73 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, second edition, 2001. 16

[3] C.F. Bornstein and J.L. Szwarcfiter. *On clique convergent graphs*. Graphs Combin. **11** (1995) 213–220. 19, 63

[4] C. Bron and J. Kerbosch. *Finding all cliques of an undirected graph–algorithm 457*. Communications of the ACM **16** (1973) 575–577. 15, 67

[5] F.F. Dragan. *Centers of graphs and the Helly property (in Russian)*. PhD thesis, Moldava State University, Chisinău, Moldava, 1989. 96

[6] F. Escalante. *Über iterierte Clique-Graphen*. Abh. Math. Sem. Univ. Hamburg **39** (1973) 59–68. 20, 67

[7] M.E. Frías-Armenta, V. Neumann-Lara and M.A. Pizaña. *Dismantlings and iterated clique graphs*. Discrete Math. **282** (2004) 263–265. 21, 68

[8] M. Frías-Armenta, F. Larrión, V. Neumann-Lara and M. Pizaña. *Edge contraction and edge removal on iterated clique graphs*. Discrete Applied Mathematics **161** (2013) 1427 – 1439. 68

[9] J. Hagauer and S. Klavzar. *Clique-gated graphs*. Discrete Mathematics **161** (1996) 143–149. 95

[10] F. Harary. *Graph theory*. Addison-Wesley Publishing Co., Reading, Mass.-Menlo Park, Calif.-London, 1969. 91

[11] M. Kahle. *Topology of random clique complexes*. Discrete Mathematics **309** (2009) 1658 – 1671. 23

[12] F. Larrión and V. Neumann-Lara. *A family of clique divergent graphs with linear growth*. Graphs Combin. **13** (1997) 263–266. 67

[13] F. Larrión and V. Neumann-Lara. *Clique divergent graphs with unbounded sequence of diameters*. Discrete Math. **197/198** (1999) 491–501. 122

[14] F. Larrión and V. Neumann-Lara. *On clique-divergent graphs with linear growth*. Discrete Math. **245** (2002) 139–153. 67

[15] F. Larrión, V. Neumann-Lara and M.A. Pizaña. *Whitney triangulations, local girth and iterated clique graphs*. Discrete Math. **258** (2002) 123–135. 96, 99

[16] F. Larrión, V. Neumann-Lara and M.A. Pizaña. *Clique divergent clockwork graphs and partial orders*. Discrete Appl. Math. **141** (2004) 195–207. 21, 67

[17] F. Larrión, V. Neumann-Lara and M.A. Pizaña. *Graph relations, clique divergence and surface triangulations*. J. Graph Theory **51** (2006) 110–122. 22, 23, 68

[18] F. Larrión, V. Neumann-Lara and M.A. Pizaña. *On expansive graphs*. European J. Combin. **30** (2009) 372–379. 23

[19] F. Larrión, M.A. Pizaña and R. Villarroel-Flores. *Random graphs, retractions and clique graphs*. Electronic Notes on Discrete Mathematics **30** (2008) 285–290. 23

[20] F. Larrión, M.A. Pizaña and R. Villarroel-Flores. *The clique behavior of circulants with three small jumps*. Ars Combinatoria **113A** (2014) 147–160. 17

[21] J.W. Moon and L. Moser. *On cliques in graphs*. Israel J. Math. **3** (1965) 23–28. 17

[22] V. Neumann-Lara. On clique-divergent graphs. In *Problèmes combinatoires et théorie des graphes (Colloq. Internat. CNRS, Univ. Orsay, Orsay, 1976)*, volume 260 of *Colloq. Internat. CNRS*, pages 313–315. CNRS, Paris, 1978. 23

[23] M.A. Pizaña. *Distances and diameters on iterated clique graphs*. Discrete Appl. Math. **141** (2004) 255–161. 19, 63

[24] E. Prisner. *Graph dynamics*. Longman, Harlow, 1995. 17

[25] M. Requardt. *(Quantum) spacetime as a statistical geometry of lumps in random networks*. Classical Quantum Gravity **17** (2000) 2029–2057. 17

[26] M. Requardt. Space-time as an order-parameter manifold in random networks and the emergence of physical points. In *Quantum theory and symmetries (Goslar, 1999)*, pages 555–561. World Sci. Publ., River Edge, NJ, 2000. 17

[27] M. Requardt. *A geometric renormalization group in discrete quantum space-time*. J. Math. Phys. **44** (2003) 5588–5615. 17

[28] J.L. Szwarcfiter. A survey on clique graphs. In B.A. Reed and C. Linhares-Sales, editors, *Recent advances in algorithms and combinatorics*, volume 11 of *CMS Books Math./Ouvrages Math. SMC*, pages 109–136. Springer, New York, 2003. 17

[29] J.L. Szwarcfiter. *Recognizing clique-Helly graphs*. Ars Combin. **45** (1997) 29–32. 96

Index