GRAPE

A Package for GAP

by

Leonard H. Soicher

School of Mathematical Sciences
Queen Mary University of London

Contents

1 Grape

This manual describes the GRAPE (Version 4.7) package for computing with graphs and groups.

GRAPE is primarily designed for the construction and analysis of finite graphs related to groups, designs, and geometries. Special emphasis is placed on the determination of regularity properties and subgraph structure. The GRAPE philosophy is that a graph *gamma* always comes together with a known subgroup G of the automorphism group of *gamma*, and that G is used to reduce the storage and CPU-time requirements for calculations with *gamma* (see [Soi93] and [Soi04]). Of course G may be the trivial group, and in this case GRAPE algorithms may perform more slowly than strictly combinatorial algorithms (although this degradation in performance is hopefully never more than a fixed constant factor).

Most GRAPE functions are written entirely in the GAP language. However, the GRAPE functions `AutomorphismGroup`, `AutGroupGraph`, `IsIsomorphicGraph`, `GraphIsomorphismClassRepresentatives`, `GraphIsomorphism` and `PartialLinearSpaces` make direct or indirect use the *nauty* [Nau90,MP14] or *bliss* [JK07] packages, via a GRAPE interface. These functions can only be used on a fully installed version of GRAPE. Installation of GRAPE is described in this chapter of the manual.

Except for the *nauty* package of B.D. Mckay included with GRAPE, the function `SmallestImageSet` by Steve Linton, the *nauty* interface by Alexander Hulpke, and the initial *bliss* interface by Jerry James, the GRAPE package was designed and written by Leonard H. Soicher, School of Mathematical Sciences, Queen Mary University of London. Except for the included *nauty* package, GRAPE is licensed under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. For details, see

`http://www.gnu.org/licenses/gpl.html` . Further licensing and copyright information for GRAPE is contained in its `COPYING` file.

If you use GRAPE to solve a problem then please send a short email about it to `L.H.Soicher@qmul.ac.uk`, and reference the GRAPE package as follows:

L.H. Soicher, The GRAPE package for GAP, Version 4.7, 2016,

`http://www.maths.qmul.ac.uk/~leonard/grape/` .

If your work made use of a function depending on *nauty* or *bliss* then you should also reference *nauty* [Nau90,MP14] or *bliss* [JK07] as appropriate.

The development of GRAPE was partially supported by a European Union HCM grant in "Computational Group Theory". It is currently partially supported by EPSRC grant EP/M022641/1 (CoDiMa: a Collaborative Computational Project in the area of Computational Discrete Mathematics).

1.1 Installing the GRAPE Package

Since version 4.5, the official GAP distribution includes the GRAPE package, which includes a 32-bit nauty/dreadnaut binary for Windows (XP and later versions). Thus, GRAPE normally requires no further installation for Windows users of GAP.

You do not need to download and unpack an archive for GRAPE unless you want to install the package separately from your main GAP installation or are installing an upgrade of GRAPE to an existing installation of GAP (see the

main GAP reference section "ref:installing a gap package"). If you do need to download GRAPE, you can find the most recent `.tar.gz` archive file for the package at

`http://www.maths.qmul.ac.uk/~leonard/grape/` , and then this archive file should be downloaded and unpacked in the `pkg` subdirectory of an appropriate GAP root directory (see the main GAP reference section "ref:gap root directories").

If your GRAPE installation does not include a pre-compiled binary of the nauty/dreadnaut programs included with GRAPE and you do not want to use an already installed version of *nauty* or *bliss*, you will need to perform compilation of the nauty/dreadnaut programs included with GRAPE, and to do this in a Unix environment, you should proceed as follows. After installing GAP, go to the GRAPE home directory (usually the directory `pkg/grape` of the GAP home directory), and run `./configure` *path*, where *path* is the path of the GAP home directory. So for example, if you install GRAPE in the `pkg` directory of the GAP home directory, run

```
./configure ../..
```

This will fetch the name of the architecture for which GAP has been most recently configured, and create a `Makefile`. Now run

```
make
```

to create the nauty/dreadnaut binary and to put it in the appropriate place. This configuration/make process for GRAPE only works for the **last** architecture for which GAP was configured. Therefore, you should always follow the above procedure to install the nauty/dreadnaut binary immediately after compiling GAP for a given configuration, say for a different architecture on a common file system. However, if you want to add GRAPE later, you can just run `./configure` again in the GAP home directory for the architecture, before performing the GRAPE configure/make process to install the nauty/dreadnaut binary for that architecture.

To use GRAPE with a separately installed version of *nauty* or *bliss* you should proceed as follows. Please note that the *nauty* interface for GRAPE has only been extensively tested with the included Version 2.2 of *nauty*, and the *bliss* interface has only been tested with Version 0.73 of *bliss*. To use a separately installed version of *nauty*, type the following commands in GAP, or place these commands in your `gaprc` file (see "ref:the gaprc file"), where `dreadnaut_or_dreadnautB_executable` should be the name of your dreadnaut or dreadnautB executable file:

```
LoadPackage("grape");
GRAPE_NAUTY := true;
GRAPE_DREADNAUT_EXE := "dreadnaut_or_dreadnautB_executable";
```

To use a separately installed version of *bliss* instead of *nauty*, type the following commands in GAP, or place these commands in your `gaprc` file (see "ref:the gaprc file"), where `bliss_executable` should be the name of your bliss executable file:

```
LoadPackage("grape");
GRAPE_NAUTY := false;
GRAPE_BLISS_EXE := "bliss_executable";
```

For example, if the bliss executable is `/usr/local/bin/bliss`, then type:

```
LoadPackage("grape");
GRAPE_NAUTY := false;
GRAPE_BLISS_EXE := "/usr/local/bin/bliss";
```

You should now test GRAPE and the interface to *nauty* or *bliss* on each architecture on which you have installed GRAPE. Start up GAP and at the prompt type

```
LoadPackage( "grape" );
```

On-line documentation for GRAPE should be available by typing

```
?GRAPE
```

Then run some tests by typing:

```
Test(Filename(DirectoriesPackageLibrary("grape","tst"),"testall.tst"));
```

This should return the value `true`.

Both dvi and pdf versions of the GRAPE manual are available (as `manual.dvi` and `manual.pdf` respectively) in the `doc` directory of the home directory of GRAPE.

If you install GRAPE, then please tell `L.H.Soicher@qmul.ac.uk`, where you should also send any comments or bug reports.

1.2 Loading GRAPE

Before using GRAPE you must load the package within GAP via:

```
gap> LoadPackage("grape");
true
```

1.3 The structure of a graph in GRAPE

In general GRAPE deals with finite directed graphs which may have loops but have no multiple edges. However, many GRAPE functions only work for **simple** graphs (i.e. no loops, and whenever $[x, y]$ is an edge then so is $[y, x]$), but these functions will check if an input graph is simple.

In GRAPE, a graph *gamma* is stored as a record, with mandatory components `isGraph`, `order`, `group`, `schreierVector`, `representatives`, and `adjacencies`. Usually, the user need not be aware of this record structure, and is strongly advised only to use GRAPE functions to construct and modify graphs.

The `order` component contains the number of vertices of *gamma*. The vertices of *gamma* are always 1,2,...,*gamma*`.order`, but they may also be given **names**, either by a user (using `AssignVertexNames`) or by a function constructing a graph (e.g. `InducedSubgraph`, `BipartiteDouble`, `QuotientGraph`). The `names` component, if present, records these names, with *gamma*`.names[`i`]` the name of vertex i. If the `names` component is not present (the user may, for example, choose to unbind it), then the names are taken to be 1,2,...,*gamma*`.order`. The `group` component records the GAP permutation group associated with *gamma* (this group must be a subgroup of the automorphism group of *gamma*). The `representatives` component records a set of orbit representatives for the action of *gamma*`.group` on the vertices of *gamma*, with *gamma*`.adjacencies[`i`]` being the set of vertices adjacent to *gamma*`.representatives[`i`]`. The `group` and `schreierVector` components are used to compute the adjacency-set of an arbitrary vertex of *gamma* (this is done by the function `Adjacency`).

The only mandatory component which may change once a graph is initially constructed is `adjacencies` (when an edge-orbit of *gamma*`.group` is added to, or removed from, *gamma*). A graph record may also have some of the optional components `isSimple`, `autGroup`, and `canonicalLabelling`, which record information about that graph.

1.4 Examples of the use of GRAPE

We give here a simple example to illustrate the use of GRAPE. All functions used are described in detail in this manual. More sophisticated examples of the use of GRAPE can be found in chapter 9, and also in the references [Cam99], [CSS99], [HL99] and [Soi06].

In the example here, we construct the Petersen graph P, and its edge graph (also called line graph) EP. We compute the global parameters of EP, and so verify that EP is distance-regular (see [BCN89]).

```
gap> LoadPackage("grape");
true
gap> P := Graph( SymmetricGroup(5), [[1,2]], OnSets,
>             function(x,y) return Intersection(x,y)=[]; end );
rec( isGraph := true, order := 10,
  group := Group([ ( 1, 2, 3, 5, 7)( 4, 6, 8, 9,10), ( 2, 4)( 6, 9)( 7,10) ]),
  schreierVector := [ -1, 1, 1, 2, 1, 1, 1, 1, 2, 2 ],
  adjacencies := [ [ 3, 5, 8 ] ], representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 2, 3 ], [ 3, 4 ], [ 1, 3 ], [ 4, 5 ], [ 2, 4 ],
      [ 1, 5 ], [ 3, 5 ], [ 1, 4 ], [ 2, 5 ] ] )
gap> Diameter(P);
2
gap> Girth(P);
5
gap> EP := EdgeGraph(P);
rec( isGraph := true, order := 15,
  group := Group([ ( 1, 4, 7, 2, 5)( 3, 6, 8, 9,12)(10,13,14,15,11),
      ( 4, 9)( 5,11)( 6,10)( 7, 8)(12,15)(13,14) ]),
  schreierVector := [ -1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 1, 2 ],
  adjacencies := [ [ 2, 3, 7, 8 ] ], representatives := [ 1 ],
  isSimple := true,
  names := [ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 1, 2 ], [ 4, 5 ] ],
      [ [ 1, 2 ], [ 3, 5 ] ], [ [ 2, 3 ], [ 4, 5 ] ], [ [ 2, 3 ], [ 1, 5 ] ],
      [ [ 2, 3 ], [ 1, 4 ] ], [ [ 3, 4 ], [ 1, 5 ] ], [ [ 3, 4 ], [ 2, 5 ] ],
      [ [ 1, 3 ], [ 4, 5 ] ], [ [ 1, 3 ], [ 2, 4 ] ], [ [ 1, 3 ], [ 2, 5 ] ],
      [ [ 2, 4 ], [ 1, 5 ] ], [ [ 2, 4 ], [ 3, 5 ] ], [ [ 3, 5 ], [ 1, 4 ] ],
      [ [ 1, 4 ], [ 2, 5 ] ] ] )
gap> GlobalParameters(EP);
[ [ 0, 0, 4 ], [ 1, 1, 2 ], [ 1, 2, 1 ], [ 4, 0, 0 ] ]
```

2 Functions to construct and modify graphs

This chapter describes the functions used to construct and modify graphs.

2.1 Graph

1 ▶ `Graph( G, L, act, rel )`
▶ `Graph( G, L, act, rel, invt )`

This is the most general and useful way of constructing a graph in GRAPE.

First suppose that the optional boolean parameter *invt* is unbound or has value `false`. Then L should be a list of elements of a set S on which the group G acts, with the action given by the function *act*. The parameter *rel* should be a boolean function defining a G-invariant relation on S (so that for g in G, x, y in S, $rel(x, y)$ if and only if $rel(act(x, g), act(y, g))$). Then the function `Graph` returns a graph *gamma* which has as vertex-names (an immutable copy of)

`Concatenation( Orbits( G, L, act ) )`

(the concatenation of the distinct orbits of the elements in L under G), and for vertices v, w of *gamma*, $[v, w]$ is an edge if and only if

rel`( VertexName( gamma, v ), VertexName( gamma, w ) )`.

Now if the parameter *invt* exists and has value `true`, then it is assumed that L is invariant under G with respect to action *act*. Then the function `Graph` behaves as above, except that the vertex-names of *gamma* become (an immutable copy of) L.

The group associated with the graph *gamma* returned is the image of G acting via *act* on *gamma*`.names`.

For example, we may use `Graph` to construct the Petersen graph as follows:

```
gap> Petersen := Graph( SymmetricGroup(5), [[1,2]], OnSets,
>                   function(x,y) return Intersection(x,y)=[]; end );
rec(
  isGraph := true,
  order := 10,
  group := Group( [ ( 1, 2, 3, 5, 7)( 4, 6, 8, 9,10), ( 2, 4)( 6, 9)( 7,10)
     ] ),
  schreierVector := [ -1, 1, 1, 2, 1, 1, 1, 1, 2, 2 ],
  adjacencies := [ [ 3, 5, 8 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 2, 3 ], [ 3, 4 ], [ 1, 3 ], [ 4, 5 ], [ 2, 4 ],
      [ 1, 5 ], [ 3, 5 ], [ 1, 4 ], [ 2, 5 ] ] )
```

The function `Graph` may be used to construct a graph in GRAPE format from an adjacency matrix for that graph. For example, suppose you have an n by n adjacency matrix A for a graph *gamma*, so that the vertex-set of *gamma*

is $\{1,\ldots,n\}$, and $[i,j]$ is an edge of *gamma* if and only if $A[i][j] = 1$. Then the graph *gamma* in GRAPE-graph format, with associated group the trivial group, is returned by `Graph( Group(()), [1..n], OnPoints, function(x,y) return A[x][y]=1; end, true );`

```
gap> A := [[0,1,0],[1,0,0],[0,0,1]];
[ [ 0, 1, 0 ], [ 1, 0, 0 ], [ 0, 0, 1 ] ]
gap> gamma := Graph( Group(()), [1..3], OnPoints,
>        function(x,y) return A[x][y]=1; end,
>        true );
rec( adjacencies := [ [ 2 ], [ 1 ], [ 3 ] ], group := Group(()),
  isGraph := true, names := [ 1, 2, 3 ], order := 3,
  representatives := [ 1, 2, 3 ], schreierVector := [ -1, -2, -3 ] )
```

2.2 EdgeOrbitsGraph

1 ▶ `EdgeOrbitsGraph( G, edges )`
▶ `EdgeOrbitsGraph( G, edges, n )`

This is a common way of constructing a graph in GRAPE.

This function returns the (directed) graph with vertex-set $\{1,\ldots,n\}$, edge-set $\cup_{e\in edges} e^G$, and associated (permutation) group G, which must act naturally on $\{1,\ldots,n\}$. The parameter *edges* should be a list of edges (lists of length 2 of vertices), although a singleton edge will be understood as an edge-list of length 1. The parameter n may be omitted, in which case n is taken to be the largest point moved by G.

Note that G may be the trivial permutation group (`Group( () )` in GAP notation), in which case the (directed) edges of *gamma* are simply those in the list *edges*.

```
gap> EdgeOrbitsGraph( Group((1,3),(1,2)(3,4)), [[1,2],[4,5]], 5 );
rec(
  isGraph := true,
  order := 5,
  group := Group( [ (1,3), (1,2)(3,4) ] ),
  schreierVector := [ -1, 2, 1, 2, -2 ],
  adjacencies := [ [ 2, 4, 5 ], [  ] ],
  representatives := [ 1, 5 ],
  isSimple := false )
```

2.3 NullGraph

1 ▶ `NullGraph( G )`
▶ `NullGraph( G, n )`

This function returns the null graph (graph with no edges) with vertex-set $\{1,\ldots,n\}$, and associated (permutation) group G. The parameter n may be omitted, in which case n is taken to be the largest point moved by G.

See also 3.20.1.

```
gap> NullGraph( Group( (1,2,3) ), 4 );
rec(
  isGraph := true,
  order := 4,
  group := Group( [ (1,2,3) ] ),
  schreierVector := [ -1, 1, 1, -2 ],
  adjacencies := [ [  ], [  ] ],
  representatives := [ 1, 4 ],
  isSimple := true )
```

2.4 CompleteGraph

1 ▶ `CompleteGraph( G )`
▶ `CompleteGraph( G, n )`
▶ `CompleteGraph( G, n, mustloops )`

This function returns the complete graph with vertex-set $\{1,\ldots,n\}$ and associated (permutation) group G. The parameter n may be omitted, in which case n is taken to be the largest point moved by G. The optional boolean parameter *mustloops* determines whether the complete graph has all loops present or no loops (default: `false` (no loops)).

See also 3.21.1.

```
gap> CompleteGraph( Group( (1,2,3), (1,2) ) );
rec(
  isGraph := true,
  order := 3,
  group := Group( [ (1,2,3), (1,2) ] ),
  schreierVector := [ -1, 1, 1 ],
  adjacencies := [ [ 2, 3 ] ],
  representatives := [ 1 ],
  isSimple := true )
```

2.5 JohnsonGraph

1 ▶ `JohnsonGraph( n, e )`

Let n and e be integers, with $n \geq e \geq 0$. Then this function returns a graph *gamma* isomorphic to the Johnson graph $J(n,e)$. The vertices (actually the vertex-names) of *gamma* are the e-subsets of $\{1,\ldots,n\}$, with x joined to y if and only if $|x \cap y| = e-1$. The group associated with *gamma* is the image of the symmetric group S_n acting on the e-subsets of $\{1,\ldots,n\}$.

```
gap> JohnsonGraph(5,3);
rec(
  isGraph := true,
  order := 10,
  group := Group( [ ( 1, 7,10, 6, 3)( 2, 8, 4, 9, 5), ( 4, 7)( 5, 8)( 6, 9)
     ] ),
  schreierVector := [ -1, 1, 1, 2, 1, 1, 1, 2, 1, 1 ],
  adjacencies := [ [ 2, 3, 4, 5, 7, 8 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 2, 5 ], [ 1, 3, 4 ], [ 1, 3, 5 ],
      [ 1, 4, 5 ], [ 2, 3, 4 ], [ 2, 3, 5 ], [ 2, 4, 5 ], [ 3, 4, 5 ] ],
  isSimple := true )
```

2.6 CayleyGraph

1 ▶ `CayleyGraph( G )`
▶ `CayleyGraph( G, gens )`
▶ `CayleyGraph( G, gens, undirected )`

Given a group G and a generating list *gens* for G, `CayleyGraph( G, gens )` returns a Cayley graph for G with respect to *gens*. The generating list *gens* is optional, and if omitted, then *gens* is taken to be `GeneratorsOfGroup( G )`. The boolean argument *undirected* is also optional, and if *undirected*=`true` (the default), then the returned graph is undirected (as if *gens* was closed under inversion, whether or not it is).

The Cayley graph *caygraph* which is returned is defined as follows: the vertices (actually the vertex-names) of *caygraph* are the elements of G; if *undirected*=`true` (the default) then vertices x, y are joined by an edge if and only if there is a g in the list *gens* with $y = gx$ or $y = g^{-1}x$; if *undirected*=`false` then $[x, y]$ is an edge if and only if there is a g in *gens* with $y = gx$.

The permutation group *caygraph*`.group` associated with *caygraph* is the image of G acting in its right regular representation.

Note It is not checked whether G is actually generated by *gens*. However, even if G is not generated by *gens*, the function still works as described above (as long as *gens* is contained in G), but returns a "Cayley graph" which is not connected.

```
gap> C:=CayleyGraph(SymmetricGroup(4),[(1,2),(2,3),(3,4)]);
rec(
  isGraph := true,
  order := 24,
  group :=
   Group( [ ( 1,10,17,19)( 2, 9,18,20)( 3,12,14,21)( 4,11,13,22)( 5, 7,16,23)
        ( 6, 8,15,24), ( 1, 7)( 2, 8)( 3, 9)( 4,10)( 5,11)( 6,12)(13,15)
        (14,16)(17,18)(19,21)(20,22)(23,24) ] ),
  schreierVector := [ -1, 1, 1, 2, 2, 1, 2, 2, 2, 1, 1, 1, 1, 2, 2, 1, 1, 2,
      1, 1, 2, 2, 1, 2 ],
  adjacencies := [ [ 2, 3, 7 ] ],
  representatives := [ 1 ],
  names := [ (), (3,4), (2,3), (2,3,4), (2,4,3), (2,4), (1,2), (1,2)(3,4),
      (1,2,3), (1,2,3,4), (1,2,4,3), (1,2,4), (1,3,2), (1,3,4,2), (1,3),
      (1,3,4), (1,3)(2,4), (1,3,2,4), (1,4,3,2), (1,4,2), (1,4,3), (1,4),
      (1,4,2,3), (1,4)(2,3) ],
  isSimple := true )
gap> Girth(C);
4
gap> Diameter(C);
6
```

2.7 AddEdgeOrbit

1 ▶ `AddEdgeOrbit(` *gamma*`,` *e* `)`
▶ `AddEdgeOrbit(` *gamma*`,` *e*`,` *H* `)`

This procedure adds the orbit of *e* under *gamma*`.group` to the edge-set of the graph *gamma*. The parameter *e* must be a sequence of length 2 of vertices of *gamma*. If the optional third parameter *H* is given then it is assumed that *e*`[2]` has the same orbit under *H* as it does under the stabilizer in *gamma*`.group` of *e*`[1]`, and this knowledge can speed up the procedure.

Note that if *gamma*`.group` is trivial then this procedure simply adds the single (directed) edge *e* to *gamma*.

See also 2.8.1.

```
gap> gamma := NullGraph( Group( (1,3), (1,2)(3,4) ) );;
gap> AddEdgeOrbit( gamma, [4,3] );
gap> gamma;
rec(
  isGraph := true,
  order := 4,
  group := Group( [ (1,3), (1,2)(3,4) ] ),
  schreierVector := [ -1, 2, 1, 2 ],
```

```
    adjacencies := [ [ 2, 4 ] ],
    representatives := [ 1 ],
    isSimple := true )
gap> GlobalParameters(gamma);
[ [ 0, 0, 2 ], [ 1, 0, 1 ], [ 2, 0, 0 ] ]
```

2.8 RemoveEdgeOrbit

1 ▶ `RemoveEdgeOrbit(` *gamma*`,` *e* `)`
▶ `RemoveEdgeOrbit(` *gamma*`,` *e*`,` *H* `)`

This procedure removes the orbit of *e* under *gamma*`.group` from the edge-set of the graph *gamma*. The parameter *e* must be a sequence of length 2 of vertices of *gamma*, but if *e* is not an edge of *gamma* then this procedure has no effect. If the optional third parameter *H* is given then it is assumed that *e*`[2]` has the same orbit under *H* as it does under the stabilizer in *gamma*`.group` of *e*`[1]`, and this knowledge can speed up the procedure.

See also 2.7.1.

```
gap> gamma := CompleteGraph( Group( (1,3), (1,2)(3,4) ) );;
gap> RemoveEdgeOrbit( gamma, [1,3] );
gap> gamma;
rec(
  isGraph := true,
  order := 4,
  group := Group( [ (1,3), (1,2)(3,4) ] ),
  schreierVector := [ -1, 2, 1, 2 ],
  adjacencies := [ [ 2, 4 ] ],
  representatives := [ 1 ],
  isSimple := true )
gap> GlobalParameters(gamma);
[ [ 0, 0, 2 ], [ 1, 0, 1 ], [ 2, 0, 0 ] ]
```

2.9 AssignVertexNames

1 ▶ `AssignVertexNames(` *gamma*`,` *names* `)`

This procedure allows the user to give new names for the vertices of *gamma*, by specifying a list *names* (of length *gamma*`.order`) of vertex-names for the vertices of *gamma*, such that *names*`[`*i*`]` contains the user's name for the *i*-th vertex of *gamma*.

An immutable copy of *names* is assigned to *gamma*`.names`.

See also 3.5.1 and 3.4.1.

```
gap> gamma := NullGraph( Group(()), 3 );
rec(
  isGraph := true,
  order := 3,
  group := Group( [ () ] ),
  schreierVector := [ -1, -2, -3 ],
  adjacencies := [ [  ], [  ], [  ] ],
  representatives := [ 1, 2, 3 ],
  isSimple := true )
gap> AssignVertexNames( gamma, ["a","b","c"] );
gap> gamma;
rec(
```

```
isGraph := true,
order := 3,
group := Group( [ () ] ),
schreierVector := [ -1, -2, -3 ],
adjacencies := [ [  ], [  ], [  ] ],
representatives := [ 1, 2, 3 ],
isSimple := true,
names := [ "a", "b", "c" ] )
```

3 Functions to inspect graphs, vertices and edges

This chapter describes functions to inspect graphs, vertices and edges.

3.1 IsGraph

1 ▶ `IsGraph( obj )`

This boolean function returns `true` if and only if *obj*, which can be an object of arbitrary type, is a graph.

```
gap> IsGraph( 1 );
false
gap> IsGraph( JohnsonGraph( 3, 2 ) );
true
```

3.2 OrderGraph

1 ▶ `OrderGraph( gamma )`

This function returns the number of vertices (the **order**) of the graph *gamma*.

```
gap> OrderGraph( JohnsonGraph( 4, 2 ) );
6
```

3.3 IsVertex

1 ▶ `IsVertex( gamma, v )`

This boolean function returns `true` if and only if *v* is vertex of the graph *gamma*.

```
gap> gamma := JohnsonGraph( 3, 2 );;
gap> IsVertex( gamma, 1 );
true
gap> IsVertex( gamma, 4 );
false
```

3.4 VertexName

1 ▶ `VertexName( gamma, v )`

This function returns (an immutable copy of) the name of vertex *v* in the graph *gamma*.

See also 3.5.1 and 2.9.1.

```
gap> VertexName( JohnsonGraph(4,2), 6 );
[ 3, 4 ]
```

3.5 VertexNames

1 ▶ `VertexNames( gamma )`

This function returns (an immutable copy of) the list of vertex-names for the graph *gamma*. The *i*-th element of this list is the name of vertex *i*.

See also 3.4.1 and 2.9.1.

```
gap> VertexNames( JohnsonGraph(4,2) );
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ]
```

3.6 Vertices

1 ▶ `Vertices( gamma )`

This function returns the vertex-set $\{1, \ldots, gamma.\texttt{order}\}$ of the graph *gamma*.

```
gap> Vertices( JohnsonGraph( 4, 2 ) );
[ 1 .. 6 ]
```

3.7 VertexDegree

1 ▶ `VertexDegree( gamma, v )`

This function returns the (out)degree of the vertex *v* of the graph *gamma*.

```
gap> VertexDegree( JohnsonGraph( 3, 2 ), 1 );
2
```

3.8 VertexDegrees

1 ▶ `VertexDegrees( gamma )`

This function returns the set of vertex (out)degrees for the graph *gamma*.

```
gap> VertexDegrees( JohnsonGraph( 4, 2 ) );
[ 4 ]
```

3.9 IsLoopy

1 ▶ `IsLoopy( gamma )`

This boolean function returns `true` if and only if the graph *gamma* has a **loop**, i.e. an edge of the form $[x, x]$.

```
gap> IsLoopy( JohnsonGraph( 4, 2 ) );
false
gap> IsLoopy( CompleteGraph( Group( (1,2,3), (1,2) ), 3 ) );
false
gap> IsLoopy( CompleteGraph( Group( (1,2,3), (1,2) ), 3, true ) );
true
```

3.10 IsSimpleGraph

1 ▶ `IsSimpleGraph( gamma )`

This boolean function returns `true` if and only if the graph *gamma* is **simple**, i.e. has no loops and whenever $[x, y]$ is an edge then so is $[y, x]$.

```
gap> IsSimpleGraph( CompleteGraph( Group( (1,2,3) ), 3 ) );
true
gap> IsSimpleGraph( CompleteGraph( Group( (1,2,3) ), 3, true ) );
false
```

3.11 Adjacency

1 ▶ `Adjacency( gamma, v )`

This function returns (a copy of) the set of vertices of the graph *gamma* adjacent to the vertex *v* of *gamma*. A vertex *w* is **adjacent** to *v* if and only if $[v, w]$ is an edge.

```
gap> Adjacency( JohnsonGraph( 4, 2 ), 1 );
[ 2, 3, 4, 5 ]
gap> Adjacency( JohnsonGraph( 4, 2 ), 6 );
[ 2, 3, 4, 5 ]
```

3.12 IsEdge

1 ▶ `IsEdge( gamma, e )`

This boolean function returns `true` if and only if *e* is an edge of the graph *gamma*.

```
gap> IsEdge( JohnsonGraph( 4, 2 ), [ 1, 2 ] );
true
gap> IsEdge( JohnsonGraph( 4, 2 ), [ 1, 6 ] );
false
```

3.13 DirectedEdges

1 ▶ `DirectedEdges( gamma )`

This function returns the set of directed (ordered) edges of the graph *gamma*.

See also 3.14.1.

```
gap> gamma := JohnsonGraph( 4, 3 );
rec( isGraph := true, order := 4, group := Group([ (1,4,3,2), (3,4) ]),
  schreierVector := [ -1, 1, 1, 1 ], adjacencies := [ [ 2, 3, 4 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 3, 4 ], [ 2, 3, 4 ] ],
  isSimple := true )
gap> DirectedEdges( gamma );
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 1 ], [ 2, 3 ], [ 2, 4 ], [ 3, 1 ],
  [ 3, 2 ], [ 3, 4 ], [ 4, 1 ], [ 4, 2 ], [ 4, 3 ] ]
gap> UndirectedEdges( gamma );
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ]
```

3.14 UndirectedEdges

1 ▶ `UndirectedEdges( gamma )`

This function returns the set of undirected (unordered) edges of *gamma*, which must be a simple graph.

See also 3.13.1.

```
gap> gamma := JohnsonGraph( 4, 3 );
rec( isGraph := true, order := 4, group := Group([ (1,4,3,2), (3,4) ]),
  schreierVector := [ -1, 1, 1, 1 ], adjacencies := [ [ 2, 3, 4 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 3, 4 ], [ 2, 3, 4 ] ],
  isSimple := true )
gap> DirectedEdges( gamma );
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 1 ], [ 2, 3 ], [ 2, 4 ], [ 3, 1 ],
  [ 3, 2 ], [ 3, 4 ], [ 4, 1 ], [ 4, 2 ], [ 4, 3 ] ]
gap> UndirectedEdges( gamma );
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ]
```

3.15 Distance

1 ▶ `Distance( gamma, X, Y )`
▶ `Distance( gamma, X, Y, G )`

This function returns the distance from X to Y in *gamma*. The parameters X and Y may be vertices or nonempty lists of vertices. We define the **distance** $d(X, Y)$ from X to Y to be the minimum length of a (directed) path joining a vertex of X to a vertex of Y if such a path exists, and -1 otherwise.

The optional parameter G, if present, is assumed to be a subgroup of Aut (*gamma*) fixing X setwise. Including such a G can speed up the function.

See also 3.16.1.

```
gap> Distance( JohnsonGraph(4,2), 1, 6 );
2
gap> Distance( JohnsonGraph(4,2), 1, 5 );
1
gap> Distance( JohnsonGraph(4,2), [1], [5,6] );
1
```

3.16 Diameter

1 ▶ `Diameter( gamma )`

This function returns the diameter of *gamma*. A diameter of -1 is returned if *gamma* is not (strongly) connected. Otherwise, the **diameter** of *gamma* is equal to the maximum (directed) distance $d(x, y)$ in *gamma* (as x and y range over all the vertices of *gamma*).

See also 3.15.1.

```
gap> Diameter( JohnsonGraph( 5, 3 ) );
2
gap> Diameter( JohnsonGraph( 5, 4 ) );
1
```

3.17 Girth

1 ► `Girth( gamma )`

This function returns the girth of *gamma*, which must be a simple graph. A girth of -1 is returned if *gamma* is a forest. Otherwise the **girth** is the length of a shortest cycle in *gamma*.

```
gap> Girth( JohnsonGraph( 4, 2 ) );
3
```

3.18 IsConnectedGraph

1 ► `IsConnectedGraph( gamma )`

This boolean function returns `true` if and only if the graph *gamma* is (strongly) **connected**, i.e. there is a (directed) path from x to y for every pair of vertices x, y of *gamma*.

```
gap> IsConnectedGraph( JohnsonGraph(4,2) );
true
gap> IsConnectedGraph( NullGraph(SymmetricGroup(4)) );
false
```

3.19 IsBipartite

1 ► `IsBipartite( gamma )`

This boolean function returns `true` if and only if the graph *gamma*, which must be simple, is **bipartite**, i.e. if the vertex-set can be expressed as the disjoint union of two sets, on each of which *gamma* induces a null graph (these two sets are called the **bicomponents** or **parts** of *gamma*).

See also 5.3.1 and 6.10.1.

```
gap> gamma := JohnsonGraph(4,2);
rec(
  isGraph := true,
  order := 6,
  group := Group( [ (1,4,6,3)(2,5), (2,4)(3,5) ] ),
  schreierVector := [ -1, 2, 1, 1, 1, 1 ],
  adjacencies := [ [ 2, 3, 4, 5 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ],
  isSimple := true )
gap> IsBipartite(gamma);
false
gap> delta := BipartiteDouble(gamma);
rec(
  isGraph := true,
  order := 12,
  group := Group( [ ( 1, 4, 6, 3)( 2, 5)( 7,10,12, 9)( 8,11),
      ( 2, 4)( 3, 5)( 8,10)( 9,11), ( 1, 7)( 2, 8)( 3, 9)( 4,10)( 5,11)
        ( 6,12) ] ),
  schreierVector := [ -1, 2, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3 ],
  adjacencies := [ [ 8, 9, 10, 11 ] ],
  representatives := [ 1 ],
  isSimple := true,
  names := [ [ [ 1, 2 ], "+" ], [ [ 1, 3 ], "+" ], [ [ 1, 4 ], "+" ],
```

```
        [ [ 2, 3 ], "+" ], [ [ 2, 4 ], "+" ], [ [ 3, 4 ], "+" ],
        [ [ 1, 2 ], "-" ], [ [ 1, 3 ], "-" ], [ [ 1, 4 ], "-" ],
        [ [ 2, 3 ], "-" ], [ [ 2, 4 ], "-" ], [ [ 3, 4 ], "-" ] ] )
gap> IsBipartite(delta);
true
```

3.20 IsNullGraph

1 ▶ `IsNullGraph(` *gamma* `)`

This boolean function returns `true` if and only if the graph *gamma* has no edges.

See also 2.3.1.

```
gap> IsNullGraph( CompleteGraph( Group(()), 3 ) );
false
gap> IsNullGraph( CompleteGraph( Group(()), 1 ) );
true
```

3.21 IsCompleteGraph

1 ▶ `IsCompleteGraph(` *gamma* `)`
▶ `IsCompleteGraph(` *gamma*`,` *mustloops* `)`

This boolean function returns `true` if and only if the graph *gamma* is a complete graph. The optional boolean parameter *mustloops* determines whether all loops must be present for *gamma* to be considered a complete graph (default: `false` (loops are ignored)).

See also 2.4.1.

```
gap> IsCompleteGraph( NullGraph( Group(()), 3 ) );
false
gap> IsCompleteGraph( NullGraph( Group(()), 1 ) );
true
gap> IsCompleteGraph( CompleteGraph(SymmetricGroup(3)), true );
false
```

4 Functions to determine regularity properties of graphs

This chapter describes functions to determine regularity properties of graphs, and a function `VertexTransitiveDRGs` which determines the distance-regular graphs on which a given transitive permutation group acts as a vertex-transitive group of automorphisms.

4.1 IsRegularGraph

1 ▶ `IsRegularGraph( gamma )`

This boolean function returns `true` if and only if the graph *gamma* is (out)regular.

```
gap> IsRegularGraph( JohnsonGraph(4,2) );
true
gap> IsRegularGraph( EdgeOrbitsGraph(Group(()),[[1,2]],2) );
false
```

4.2 LocalParameters

1 ▶ `LocalParameters( gamma, V )`
▶ `LocalParameters( gamma, V, G )`

Let *gamma* be a simple connected graph. Then this function determines all local parameters $c_i(V)$, $a_i(V)$, and $b_i(V)$ that *gamma* may have, with respect to the singleton vertex or nonempty list of vertices V. We say that *gamma* has the **local parameter** $c_i(V)$ (respectively $a_i(V)$, $b_i(V)$), with respect to V, if the number of vertices at distance $i-1$ (respectively i, $i+1$) from V that are adjacent to a vertex w at distance i from V (see 3.15.1) is the constant $c_i(V)$ (respectively $a_i(V)$, $b_i(V)$) depending only on i and V (and not w).

The function `LocalParameters` returns a list whose i-th element is the list $[c_{i-1}(V), a_{i-1}(V), b_{i-1}(V)]$, except that if some local parameter does not exist then -1 is put in its place.

This function can be used to determine whether a given subset of the vertices of a graph is a distance-regular code in that graph.

The optional parameter G, if present, is assumed to be a subgroup of Aut (*gamma*) fixing V setwise. Including such a G can speed up the function.

```
gap> gamma := JohnsonGraph(4,2);;
gap> LocalParameters( gamma, 1 );
[ [ 0, 0, 4 ], [ 1, 2, 1 ], [ 4, 0, 0 ] ]
gap> LocalParameters( gamma, [1,6] );
[ [ 0, 0, 4 ], [ 2, 2, 0 ] ]
gap> LocalParameters( gamma, [1,2] );
[ [ 0, 1, 3 ], [ -1, -1, 0 ] ]
```

4.3 GlobalParameters

1 ▶ `GlobalParameters( gamma )`

Let *gamma* be a simple connected graph, and $0 \leq i \leq$ `Diameter`$(gamma)$. This function determines all global parameters c_i, a_i, and b_i that *gamma* may have. We say that *gamma* has the **global parameter** c_i (respectively a_i, b_i) if the number of vertices at distance $i-1$ (respectively i, $i+1$) from a vertex v that are adjacent to a vertex w at distance i from v is the constant c_i (respectively a_i, b_i) depending only on i (and not v and w).

The function `GlobalParameters` returns a list of length `Diameter`$(gamma)$+1, whose i-th element is the list $[c_{i-1}, a_{i-1}, b_{i-1}]$, except that if some global parameter does not exist then -1 is put in its place.

Note that *gamma* is **distance-regular** if and only if this function returns no -1 in place of a global parameter (see [BCN89]).

See also 4.2.1 and 4.4.1.

```
gap> gamma := JohnsonGraph(4,2);;
gap> GlobalParameters( gamma );
[ [ 0, 0, 4 ], [ 1, 2, 1 ], [ 4, 0, 0 ] ]
gap> GlobalParameters( BipartiteDouble(gamma) );
[ [ 0, 0, 4 ], [ 1, 0, 3 ], [ -1, 0, -1 ], [ 4, 0, 0 ] ]
```

4.4 IsDistanceRegular

1 ▶ `IsDistanceRegular( gamma )`

This boolean function returns `true` if and only if *gamma* is **distance-regular**, i.e. *gamma* is simple, connected, and all global parameters c_i, a_i, b_i exist for $0 \leq i \leq$ `Diameter`$(gamma)$ (see [BCN89]).

See also 4.3.1.

```
gap> gamma := JohnsonGraph(4,2);;
gap> IsDistanceRegular( gamma );
true
gap> IsDistanceRegular( BipartiteDouble(gamma) );
false
```

4.5 CollapsedAdjacencyMat

1 ▶ `CollapsedAdjacencyMat( gamma )`
▶ `CollapsedAdjacencyMat( G, gamma )`

The second form of this function returns the collapsed adjacency matrix for *gamma*, where the collapsing group is G. It is assumed that G is a subgroup of Aut$(gamma)$.

The (i,j)-entry of the collapsed adjacency matrix equals the number of edges in $\{[x,y] \mid y \in j\text{-th } G\text{-orbit}\}$, where x is a fixed vertex in the i-th G-orbit.

In the case where this function is given just one argument, then it must be a graph *gamma* with the property that *gamma*`.group` is transitive on the vertex-set of *gamma*. In this case, the returned collapsed adjacency matrix for *gamma* is with respect to the stabilizer in *gamma*`.group` of 1.

The reader is warned that collapsed adjacency matrices can have different, but related meanings depending on the setting and the author.

See also 4.6.1.

```
gap> gamma := JohnsonGraph(4,2);
rec( isGraph := true, order := 6,
  group := Group([ (1,4,6,3)(2,5), (2,4)(3,5) ]),
  schreierVector := [ -1, 2, 1, 1, 1, 1 ], adjacencies := [ [ 2, 3, 4, 5 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ],
  isSimple := true )
gap> G := Stabilizer( gamma.group, [1,6], OnSets );;
gap> CollapsedAdjacencyMat( G, gamma );
[ [ 0, 4 ], [ 2, 2 ] ]
gap> CollapsedAdjacencyMat( gamma );
[ [ 0, 4, 0 ], [ 1, 2, 1 ], [ 0, 4, 0 ] ]
```

4.6 OrbitalGraphColadjMats

1 ▶ `OrbitalGraphColadjMats( G )`
▶ `OrbitalGraphColadjMats( G, H )`

This function returns a list of collapsed adjacency matrices for the orbital digraphs of the transitive permutation group G, collapsed with respect to `Stabilizer(`G`,1)` (creating collapsed adjacency matrices for the orbital digraphs in the sense of [PS97]). Also, the matrices are collapsed with respect to a fixed ordering of the orbits of `Stabilizer(`G`,1)`, with the trivial orbit `[1]` coming first.

The optional parameter H, if included, should be equal to `Stabilizer(`G`,1)`. The knowledge of this stabilizer can speed up the function.

The reader is warned that collapsed adjacency matrices can have different, but related meanings depending on the setting and the author.

See also 4.5.1.

```
gap> OrbitalGraphColadjMats( SymmetricGroup(7) );
[ [ [ 1, 0 ], [ 0, 1 ] ], [ [ 0, 6 ], [ 1, 5 ] ] ]
```

4.7 VertexTransitiveDRGs

1 ▶ `VertexTransitiveDRGs(` *coladjmats* `)`
▶ `VertexTransitiveDRGs(` G `)`

This function can determine (among other things) all the distance-regular graphs on which a given transitive permutation group G acts as a vertex-transitive group of automorphisms (as long as the permutation rank of G is not too large).

In the first form of this function, the input parameter *coladjmats* must be a list of collapsed adjacency matrices for the orbital digraphs of some transitive permutation group G, collapsed with respect to a point stabilizer (such as the list of matrices produced by the function `OrbitalGraphColadjMats`). It is assumed that the orbital/suborbit indexing used is the same as that for the rows (and columns) of each of the matrices, as well as for the indexing of the matrices themselves, with the trivial orbital first, so that, in particular, *coladjmats*`[1]` must be an identity matrix.

In the second form of this function, the input parameter G must be a transitive permutation group, and then the result returned will be the same as `VertexTransitiveDRGs( OrbitalGraphColadjMats(` G `) )`.

In either case, this function returns a record *result*, which gives information on the transitive group G acting on its natural set V. The most important component of this record is the list `orbitalCombinations`, whose elements give the sets of (the indices of) the G-orbitals whose union gives the edge-set of a distance-regular graph with vertex-set V. The component `intersectionArrays` gives the corresponding intersection arrays. The component `degree` is the

degree of the permutation group G, `rank` is its (permutation) rank, and `isPrimitive` is true if G is primitive, and `false` otherwise.

The techniques used in this function and definitions of the terms used above can be found in [PS97].

Warning This function checks all subsets of `[2..`*result*`.rank]`, so the permutation rank of G must not be large!

```
gap> m22:=PrimitiveGroup(22,1);;
gap> syl:=SylowSubgroup(m22,11);;
gap> part:=Set(Orbit(syl,1));;
gap> l211:=Stabilizer(m22,part,OnSets);;
gap> rt:=RightTransversal(m22,l211);;
gap> m22big:=Action(m22,rt,OnRight);;
gap> v:=VertexTransitiveDRGs(m22big);
rec( degree := 672, rank := 6, isPrimitive := true,
  orbitalCombinations := [ [ 2, 3, 4, 5, 6 ], [ 2, 4 ], [ 3, 5, 6 ], [ 3, 6 ]
     ],
  intersectionArrays := [ [ [ 0, 0, 671 ], [ 1, 670, 0 ] ], [ [ 0, 0, 495 ],
          [ 1, 366, 128 ], [ 360, 135, 0 ] ],
      [ [ 0, 0, 176 ], [ 1, 40, 135 ], [ 48, 128, 0 ] ],
      [ [ 0, 0, 110 ], [ 1, 28, 81 ], [ 18, 80, 12 ], [ 90, 20, 0 ] ] ] )
```

5 Some special vertex subsets of a graph

This chapter describes functions to determine certain special vertex subsets of a graph.

5.1 ConnectedComponent

1 ▶ `ConnectedComponent( gamma, v )`

This function returns the set of all vertices in *gamma* which can be reached by a path starting at the vertex *v*. The graph *gamma* must be simple.

See also 5.2.1.

```
gap> ConnectedComponent( NullGraph( Group((1,2)) ), 2 );
[ 2 ]
gap> ConnectedComponent( JohnsonGraph(4,2), 2 );
[ 1, 2, 3, 4, 5, 6 ]
```

5.2 ConnectedComponents

1 ▶ `ConnectedComponents( gamma )`

This function returns a list of the vertex sets of the connected components of *gamma*, which must be a simple graph.

See also 5.1.1.

```
gap> ConnectedComponents( NullGraph( Group((1,2,3,4)) ) );
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ] ]
gap> ConnectedComponents( JohnsonGraph(4,2) );
[ [ 1, 2, 3, 4, 5, 6 ] ]
```

5.3 Bicomponents

1 ▶ `Bicomponents( gamma )`

If the graph *gamma*, which must be simple, is bipartite, this function returns a length 2 list of bicomponents or parts of *gamma*, otherwise the empty list is returned.

Note If *gamma* is bipartite but not connected, then its set of bicomponents is not uniquely determined.

See also 3.19.1.

```
gap> Bicomponents( NullGraph(SymmetricGroup(4)) );
[ [ 1 .. 3 ], [ 4 ] ]
gap> Bicomponents( JohnsonGraph(4,2) );
[  ]
gap> Bicomponents( BipartiteDouble( JohnsonGraph(4,2) ) );
[ [ 1, 2, 3, 4, 5, 6 ], [ 7, 8, 9, 10, 11, 12 ] ]
```

5.4 DistanceSet

1 ▶ `DistanceSet( gamma, distances, V )`
▶ `DistanceSet( gamma, distances, V, G )`

Let V be a vertex or a nonempty list of vertices of *gamma*. This function returns the set of vertices w of *gamma*, such that $d(V,w)$ is in *distances* (a list or singleton distance).

The optional parameter G, if present, is assumed to be a subgroup of Aut$(gamma)$ fixing V setwise. Including such a G can speed up the function.

See also 3.15.1 and 6.2.1.

```
gap> DistanceSet( JohnsonGraph(4,2), 1, [1,6] );
[ 2, 3, 4, 5 ]
```

5.5 Layers

1 ▶ `Layers( gamma, V )`
▶ `Layers( gamma, V, G )`

Let V be a vertex or a nonempty list of vertices of *gamma*. This function returns a list whose i-th element is the set of vertices of *gamma* at distance $i-1$ from V.

The optional parameter G, if present, is assumed to be a subgroup of Aut$(gamma)$ which fixes V setwise. Including such a G can speed up the function.

See also 3.15.1.

```
gap> Layers( JohnsonGraph(4,2), 6 );
[ [ 6 ], [ 2, 3, 4, 5 ], [ 1 ] ]
```

5.6 IndependentSet

1 ▶ `IndependentSet( gamma )`
▶ `IndependentSet( gamma, indset )`
▶ `IndependentSet( gamma, indset, forbidden )`

Returns a (hopefully large) independent set of the graph *gamma*, which must be simple. An **independent set** of *gamma* is a set of vertices of *gamma*, no two of which are joined by an edge. At present, a greedy algorithm is used. The returned independent set will contain the (assumed) independent set *indset* (default: `[]`), and not contain any element of *forbidden* (default: `[]`, in which case the returned independent set is maximal).

An error is signalled if *indset* and *forbidden* have non-trivial intersection.

See also 7.2.1 and 7.3.1, which can be used on the complement graph of *gamma* to look seriously for independent sets.

```
gap> IndependentSet( JohnsonGraph(4,2), [3] );
[ 3, 4 ]
```

6 Functions to construct new graphs from old

This chapter describes functions to construct new graphs from old ones.

6.1 InducedSubgraph

1 ▶ `InducedSubgraph(` *gamma*`,` *V* `)`
▶ `InducedSubgraph(` *gamma*`,` *V*`,` *G* `)`

This function returns the subgraph of *gamma* induced on the vertex list *V* (which must not contain repeated elements). If the optional third parameter *G* is given, then it is assumed that *G* fixes *V* setwise, and is a group of automorphisms of the induced subgraph when restricted to *V*. In that case, the image of *G* acting on *V* is the group associated with the induced subgraph. If no such *G* is given then the associated group is trivial. The name of vertex *i* in the induced subgraph is equal to the name of vertex *V*[*i*] in *gamma*.

```
gap> gamma := JohnsonGraph(4,2);;
gap> S := [2,3,4,5];;
gap> square := InducedSubgraph( gamma, S, Stabilizer(gamma.group,S,OnSets) );
rec(
  isGraph := true,
  order := 4,
  group := Group( [ (1,4), (1,3)(2,4), (1,2)(3,4) ] ),
  schreierVector := [ -1, 3, 2, 1 ],
  adjacencies := [ [ 2, 3 ] ],
  representatives := [ 1 ],
  isSimple := true,
  names := [ [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ] ] )
gap> GlobalParameters(square);
[ [ 0, 0, 2 ], [ 1, 0, 1 ], [ 2, 0, 0 ] ]
```

6.2 DistanceSetInduced

1 ▶ `DistanceSetInduced(` *gamma*`,` *distances*`,` *V* `)`
▶ `DistanceSetInduced(` *gamma*`,` *distances*`,` *V*`,` *G* `)`

Let *V* be a vertex or a nonempty list of vertices of *gamma*. This function returns the subgraph of *gamma* induced on the set of vertices *w* of *gamma* such that $d(V, w)$ is in *distances* (a list or singleton distance).

The optional parameter *G*, if present, is assumed to be a subgroup of Aut (*gamma*) fixing *V* setwise. Including such a *G* can speed up the function.

See also 3.15.1 and 5.4.1.

```
gap> DistanceSetInduced( JohnsonGraph(4,2), [0,1], [1] );
rec(
  isGraph := true,
  order := 5,
  group := Group( [ (2,3)(4,5), (2,5)(3,4) ] ),
  schreierVector := [ -1, -2, 1, 2, 2 ],
  adjacencies := [ [ 2, 3, 4, 5 ], [ 1, 3, 4 ] ],
  representatives := [ 1, 2 ],
  isSimple := true,
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ] ] )
```

6.3 DistanceGraph

1 ► `DistanceGraph(` *gamma*, *distances* `)`

This function returns the graph *delta*, with the same vertex-set (and vertex-names) as *gamma*, such that $[x, y]$ is an edge of *delta* if and only if $d(x, y)$ (in *gamma*) is in *distances* (a list or singleton distance).

```
gap> DistanceGraph( JohnsonGraph(4,2), [2] );
rec(
  isGraph := true,
  order := 6,
  group := Group( [ (1,4,6,3)(2,5), (2,4)(3,5) ] ),
  schreierVector := [ -1, 2, 1, 1, 1, 1 ],
  adjacencies := [ [ 6 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ],
  isSimple := true )
gap> ConnectedComponents(last);
[ [ 1, 6 ], [ 2, 5 ], [ 3, 4 ] ]
```

6.4 ComplementGraph

1 ► `ComplementGraph(` *gamma* `)`
► `ComplementGraph(` *gamma*, *comploops* `)`

This function returns the complement of the graph *gamma*. The optional boolean parameter *comploops* determines whether or not loops/nonloops are complemented (default: `false` (loops/nonloops are not complemented)). The returned graph will have the same vertex-names as *gamma*.

```
gap> ComplementGraph( NullGraph(SymmetricGroup(3)) );
rec(
  isGraph := true,
  order := 3,
  group := SymmetricGroup( [ 1 .. 3 ] ),
  schreierVector := [ -1, 1, 1 ],
  adjacencies := [ [ 2, 3 ] ],
  representatives := [ 1 ],
  isSimple := true )
gap> IsLoopy(last);
false
gap> IsLoopy(ComplementGraph(NullGraph(SymmetricGroup(3)),true));
true
```

6.5 PointGraph

1 ▶ `PointGraph( gamma )`
▶ `PointGraph( gamma, v )`

Assuming that *gamma* is simple, connected, and bipartite, this function returns the induced subgraph on the connected component of `DistanceGraph(`*gamma*`,2)` containing the vertex v (default: $v = 1$).

Thus, if *gamma* is the incidence graph of a connected geometry of rank 2, and v represents a point, then the point graph of the geometry is returned.

```
gap> BipartiteDouble( CompleteGraph(SymmetricGroup(4)) );;
gap> PointGraph(last);
rec(
  isGraph := true,
  order := 4,
  group := Group( [ (1,2), (1,2,3,4) ] ),
  schreierVector := [ -1, 1, 2, 2 ],
  adjacencies := [ [ 2, 3, 4 ] ],
  representatives := [ 1 ],
  isSimple := true,
  names := [ [ 1, "+" ], [ 2, "+" ], [ 3, "+" ], [ 4, "+" ] ] )
gap> IsCompleteGraph(last);
true
```

6.6 EdgeGraph

1 ▶ `EdgeGraph( gamma )`

This function return a graph isomorphic to the the edge graph (also called the line graph) of the simple graph *gamma*.

This **edge graph** *delta* has the unordered edges of *gamma* as vertices, and e is joined to f in *delta* precisely when $|e \cap f| = 1$. The name of the vertex of the returned graph corresponding to the unordered edge $[v, w]$ of *gamma* (with $v < w$) is `[VertexName(`*gamma*`,`v`),VertexName(`*gamma*`,`w`)]`.

```
gap> EdgeGraph( CompleteGraph(SymmetricGroup(5)) );
rec(
  isGraph := true,
  order := 10,
  group := Group( [ ( 1, 5, 8,10, 4)( 2, 6, 9, 3, 7), ( 2, 5)( 3, 6)( 4, 7)
     ] ),
  schreierVector := [ -1, 2, 2, 1, 1, 1, 2, 1, 1, 1 ],
  adjacencies := [ [ 2, 3, 4, 5, 6, 7 ] ],
  representatives := [ 1 ],
  isSimple := true,
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ], [ 2, 4 ],
      [ 2, 5 ], [ 3, 4 ], [ 3, 5 ], [ 4, 5 ] ] )
gap> GlobalParameters(last);
[ [ 0, 0, 6 ], [ 1, 3, 2 ], [ 4, 2, 0 ] ]
```

6.7 SwitchedGraph

1 ▶ `SwitchedGraph( gamma, V )`
▶ `SwitchedGraph( gamma, V, H )`

This function returns the switched graph *delta* of the graph *gamma*, switched with respect to the vertex list (or singleton vertex) V.

The returned graph *delta* has vertex-set (and vertex-names) the same as *gamma*. If vertices x, y of *delta* are both in V or both not in V, then $[x, y]$ is an edge of *delta* if and only if $[x, y]$ is an edge of *gamma*; otherwise $[x, y]$ is an edge of *delta* if and only if $[x, y]$ is not an edge of *gamma*. If the optional third argument H is given, then it is assumed to be a subgroup of Aut(*gamma*) stabilizing V setwise.

```
gap> J:=JohnsonGraph(4,2);
rec(
  isGraph := true,
  order := 6,
  group := Group( [ (1,4,6,3)(2,5), (2,4)(3,5) ] ),
  schreierVector := [ -1, 2, 1, 1, 1, 1 ],
  adjacencies := [ [ 2, 3, 4, 5 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ],
  isSimple := true )
gap> S:=SwitchedGraph(J,[1,6]);
rec(
  isGraph := true,
  order := 6,
  group := Group( () ),
  schreierVector := [ -1, -2, -3, -4, -5, -6 ],
  adjacencies := [ [  ], [ 3, 4 ], [ 2, 5 ], [ 2, 5 ], [ 3, 4 ], [  ] ],
  representatives := [ 1, 2, 3, 4, 5, 6 ],
  isSimple := true,
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ] )
gap> ConnectedComponents(S);
[ [ 1 ], [ 2, 3, 4, 5 ], [ 6 ] ]
```

6.8 UnderlyingGraph

1 ▶ `UnderlyingGraph( gamma )`

This function returns the underlying graph *delta* of *gamma*. The graph *delta* has the same vertex-set (and vertex-names) as *gamma*, and has an edge $[x, y]$ precisely when *gamma* has an edge $[x, y]$ or an edge $[y, x]$. This function also sets the `isSimple` components of *gamma* and *delta*.

```
gap> gamma := EdgeOrbitsGraph( Group((1,2,3,4)), [1,2] );
rec(
  isGraph := true,
  order := 4,
  group := Group( [ (1,2,3,4) ] ),
  schreierVector := [ -1, 1, 1, 1 ],
  adjacencies := [ [ 2 ] ],
  representatives := [ 1 ],
  isSimple := false )
gap> UnderlyingGraph(gamma);
rec(
```

```
  isGraph := true,
  order := 4,
  group := Group( [ (1,2,3,4) ] ),
  schreierVector := [ -1, 1, 1, 1 ],
  adjacencies := [ [ 2, 4 ] ],
  representatives := [ 1 ],
  isSimple := true )
```

6.9 QuotientGraph

1 ▶ `QuotientGraph(` *gamma*`,` *R* `)`

Let *S* be the smallest *gamma*`.group`-invariant equivalence relation on the vertices of *gamma*, such that *S* contains the relation *R* (which should be a list of ordered pairs (length 2 lists) of vertices of *gamma*). Then this function returns a graph isomorphic to the quotient *delta* of the graph *gamma*, defined as follows. The vertices of *delta* are the equivalence classes of *S*, and $[X, Y]$ is an edge of *delta* if and only if $[x, y]$ is an edge of *gamma* for some $x \in X$, $y \in Y$. The name of a vertex *v* in the returned graph is a list (not necessarily ordered) of the vertex-names of *gamma* for the vertices in the equivalence class corresponding to *v*.

```
gap> gamma := JohnsonGraph(4,2);;
gap> QuotientGraph( gamma, [[1,6]] );
rec(
  isGraph := true,
  order := 3,
  group := Group( [ (1,3), (2,3) ] ),
  schreierVector := [ -1, 2, 1 ],
  adjacencies := [ [ 2, 3 ] ],
  representatives := [ 1 ],
  isSimple := true,
  names := [ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 1, 3 ], [ 2, 4 ] ],
      [ [ 1, 4 ], [ 2, 3 ] ] ] )
gap> IsCompleteGraph(last);
true
```

6.10 BipartiteDouble

1 ▶ `BipartiteDouble(` *gamma* `)`

This function returns the bipartite double of the graph *gamma*, as defined in [BCN89].

```
gap> gamma := JohnsonGraph(4,2);;
gap> IsBipartite(gamma);
false
gap> delta := BipartiteDouble(gamma);
rec(
  isGraph := true,
  order := 12,
  group := Group( [ ( 1, 4, 6, 3)( 2, 5)( 7,10,12, 9)( 8,11),
      ( 2, 4)( 3, 5)( 8,10)( 9,11), ( 1, 7)( 2, 8)( 3, 9)( 4,10)( 5,11)
        ( 6,12) ] ),
  schreierVector := [ -1, 2, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3 ],
  adjacencies := [ [ 8, 9, 10, 11 ] ],
  representatives := [ 1 ],
  isSimple := true,
```

```
    names := [ [ [ 1, 2 ], "+" ], [ [ 1, 3 ], "+" ], [ [ 1, 4 ], "+" ],
        [ [ 2, 3 ], "+" ], [ [ 2, 4 ], "+" ], [ [ 3, 4 ], "+" ],
        [ [ 1, 2 ], "-" ], [ [ 1, 3 ], "-" ], [ [ 1, 4 ], "-" ],
        [ [ 2, 3 ], "-" ], [ [ 2, 4 ], "-" ], [ [ 3, 4 ], "-" ] ] )
gap> IsBipartite(delta);
true
```

6.11 GeodesicsGraph

1 ► `GeodesicsGraph( gamma, x, y )`

This function returns the the graph induced on the set of geodesics in *gamma* between the vertices *x* and *y*, but including neither *x* nor *y*. This function is only for a simple graph *gamma*.

```
gap> GeodesicsGraph( JohnsonGraph(4,2), 1, 6 );
rec(
  isGraph := true,
  order := 4,
  group := Group( [ (1,3)(2,4), (1,4)(2,3), (2,3) ] ),
  schreierVector := [ -1, 2, 1, 2 ],
  adjacencies := [ [ 2, 3 ] ],
  representatives := [ 1 ],
  isSimple := true,
  names := [ [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ] ] )
gap> GlobalParameters(last);
[ [ 0, 0, 2 ], [ 1, 0, 1 ], [ 2, 0, 0 ] ]
```

6.12 CollapsedIndependentOrbitsGraph

1 ► `CollapsedIndependentOrbitsGraph( G, gamma )`
► `CollapsedIndependentOrbitsGraph( G, gamma, N )`

Given a subgroup *G* of the automorphism group of the simple graph *gamma*, this function returns a graph isomorphic to *delta*, defined as follows. The vertices of *delta* are those *G*-orbits of the vertices of *gamma* that are independent sets in *gamma*, and *x* is joined to *y* in *delta* if and only if $x \cup y$ is **not** an independent set in *gamma*. The name of a vertex *v* in the returned graph is a list (not necessarily ordered) of the vertex-names of *gamma* for the vertices in the *G*-orbit corresponding to *v*.

If the optional parameter *N* is given, then it is assumed to be a subgroup of $\mathrm{Aut}(gamma)$ preserving the set of *G*-orbits of the vertices of *gamma* (for example, the normalizer in *gamma*`.group` of *G*). This information can make the function more efficient.

```
gap> G := Group( (1,2) );;
gap> gamma := NullGraph( SymmetricGroup(3) );;
gap> CollapsedIndependentOrbitsGraph( G, gamma );
rec(
  isGraph := true,
  order := 2,
  group := Group( [ () ] ),
  schreierVector := [ -1, -2 ],
  adjacencies := [ [  ], [  ] ],
  representatives := [ 1, 2 ],
  isSimple := true,
  names := [ [ 1, 2 ], [ 3 ] ] )
```

```
gap> gamma := CompleteGraph( SymmetricGroup(3) );;
gap> CollapsedIndependentOrbitsGraph( G, gamma );
rec(
  isGraph := true,
  order := 1,
  group := Group( [ () ] ),
  schreierVector := [ -1 ],
  adjacencies := [ [  ] ],
  representatives := [ 1 ],
  isSimple := true,
  names := [ [ 3 ] ] )
```

6.13 CollapsedCompleteOrbitsGraph

1 ► `CollapsedCompleteOrbitsGraph( G, gamma )`
► `CollapsedCompleteOrbitsGraph( G, gamma, N )`

Given a subgroup G of the automorphism group of the simple graph *gamma*, this function returns a graph isomorphic to *delta*, defined as follows. The vertices of *delta* are those G-orbits of the vertices of *gamma* on which complete subgraphs are induced in *gamma*, and x is joined to y in *delta* if and only if $x \neq y$ and the subgraph of *gamma* induced on $x \cup y$ is a complete graph. The name of a vertex v in the returned graph is a list (not necessarily ordered) of the vertex-names of *gamma* for the vertices in the G-orbit corresponding to v.

If the optional parameter N is given, then it is assumed to be a subgroup of $\mathrm{Aut}(gamma)$ preserving the set of G-orbits of the vertices of *gamma* (for example, the normalizer in *gamma*`.group` of G). This information can make the function more efficient.

```
gap> G := Group( (1,2) );;
gap> gamma := NullGraph( SymmetricGroup(3) );;
gap> CollapsedCompleteOrbitsGraph( G, gamma );
rec(
  isGraph := true,
  order := 1,
  group := Group( [ () ] ),
  schreierVector := [ -1 ],
  adjacencies := [ [  ] ],
  representatives := [ 1 ],
  names := [ [ 3 ] ],
  isSimple := true )
gap> gamma := CompleteGraph( SymmetricGroup(3) );;
gap> CollapsedCompleteOrbitsGraph( G, gamma );
rec(
  isGraph := true,
  order := 2,
  group := Group( [ () ] ),
  schreierVector := [ -1, -2 ],
  adjacencies := [ [ 2 ], [ 1 ] ],
  representatives := [ 1, 2 ],
  names := [ [ 1, 2 ], [ 3 ] ],
  isSimple := true )
```

6.14 NewGroupGraph

1 ► `NewGroupGraph( G, gamma )`

This function returns a copy *delta* of *gamma*, except that the group associated with *delta* is *G*, which is assumed to be a subgroup of Aut (*delta*).

Note that the results of some functions of a graph depend on the group associated with that graph (which must always be a subgroup of the automorphism group of the graph).

```
gap> gamma := JohnsonGraph(4,2);;
gap> aut := AutGroupGraph(gamma);
Group([ (3,4), (2,3)(4,5), (1,2)(5,6) ])
gap> Size(gamma.group);
24
gap> Size(aut);
48
gap> delta := NewGroupGraph( aut, gamma );;
gap> Size(delta.group);
48
gap> IsIsomorphicGraph( gamma, delta );
true
```

7 Vertex-Colouring and Complete Subgraphs

The following sections describe functions for (proper) vertex-colouring or determining complete subgraphs of given graphs. The function `CompleteSubgraphsOfGivenSize` can also be used to determine the complete subgraphs with given vertex-weight sum in a vertex-weighted graph, where the weights can be positive integers or non-zero vectors of non-negative integers.

7.1 VertexColouring

1 ▶ `VertexColouring(` *gamma* `)`

This function returns a proper vertex-colouring C for the graph *gamma*, which must be simple.

This **proper vertex-colouring** C is a list of positive integers (the **colours**), indexed by the vertices of *gamma*, with the property that $C[i] \neq C[j]$ whenever $[i,j]$ is an edge of *gamma*. At present a greedy algorithm is used, and the number of colours used is by no means guaranteed to be minimal.

```
gap> VertexColouring( JohnsonGraph(4,2) );
[ 1, 3, 2, 2, 3, 1 ]
```

7.2 CompleteSubgraphs

1 ▶ `CompleteSubgraphs(` *gamma* `)`
▶ `CompleteSubgraphs(` *gamma*`,` *k* `)`
▶ `CompleteSubgraphs(` *gamma*`,` *k*`,` *alls* `)`

Let *gamma* be a simple graph and k an integer. This function returns a set K of complete subgraphs of *gamma*, where a complete subgraph is represented by its vertex-set. If k is non-negative then the elements of K each have size k, otherwise the elements of K represent maximal complete subgraphs of *gamma*. (A **maximal complete subgraph** of *gamma* is a complete subgraph of *gamma* which is not properly contained in another complete subgraph of *gamma*.) The default for k is -1, i.e. maximal complete subgraphs. See also `CompleteSubgraphsOfGivenSize`, which can be used to compute the maximal complete subgraphs of given size, and can also be used to determine the (maximal or otherwise) complete subgraphs with given vertex-weight sum in a vertex-weighted graph.

The optional parameter *alls* controls how many complete subgraphs are returned. The valid values for *alls* are 0, 1 (the default), and 2.

Warning: Using the default value of 1 for *alls* (see below) means that more than one element may be returned for some *gamma*`.group` orbit(s) of the required complete subgraphs. To obtain just one element from each *gamma*`.group` orbit of the required complete subgraphs, you must give the value 2 to the parameter *alls*.

If *alls*=0 (or `false` for backward compatibility) then K will contain at most one element. In this case, if k is negative then K will contain just one maximal complete subgraph, and if k is non-negative then K will contain a complete subgraph of size k if and only if such a subgraph is contained in *gamma*.

If *alls*=1 (or `true` for backward compatibility) then K will contain (perhaps properly) a set of *gamma*`.group` orbit-representatives of the maximal (if k is negative) or size k (if k is non-negative) complete subgraphs of *gamma*.

If *alls*=2 then K will be a set of *gamma*`.group` orbit-representatives of the maximal (if k is negative) or size k (if k is non-negative) complete subgraphs of *gamma*. This option can be more costly than when *alls*=1.

Before applying `CompleteSubgraphs`, one may want to associate the full automorphism group of *gamma* with *gamma*, via *gamma* `:= NewGroupGraph( AutGroupGraph(`*gamma*`),` *gamma* `);`.

An alternative name for this function is `Cliques` .

See also 7.3.1.

```
gap> gamma := JohnsonGraph(5,2);
rec( isGraph := true, order := 10,
  group := Group([ ( 1, 5, 8,10, 4)( 2, 6, 9, 3, 7), ( 2, 5)( 3, 6)( 4, 7) ]),
  schreierVector := [ -1, 2, 2, 1, 1, 1, 2, 1, 1, 1 ],
  adjacencies := [ [ 2, 3, 4, 5, 6, 7 ] ], representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ], [ 2, 4 ],
      [ 2, 5 ], [ 3, 4 ], [ 3, 5 ], [ 4, 5 ] ], isSimple := true )
gap> CompleteSubgraphs(gamma);
[ [ 1, 2, 3, 4 ], [ 1, 2, 5 ] ]
gap>  CompleteSubgraphs(gamma,3,2);
[ [ 1, 2, 3 ], [ 1, 2, 5 ] ]
gap> CompleteSubgraphs(gamma,-1,0);
[ [ 1, 2, 5 ] ]
```

7.3 CompleteSubgraphsOfGivenSize

1 ▶ `CompleteSubgraphsOfGivenSize(` *gamma*`,` *k* `)`
▶ `CompleteSubgraphsOfGivenSize(` *gamma*`,` *k*`,` *alls* `)`
▶ `CompleteSubgraphsOfGivenSize(` *gamma*`,` *k*`,` *alls*`,` *maxi* `)`
▶ `CompleteSubgraphsOfGivenSize(` *gamma*`,` *k*`,` *alls*`,` *maxi*`,` *col* `)`
▶ `CompleteSubgraphsOfGivenSize(` *gamma*`,` *k*`,` *alls*`,` *maxi*`,` *col*`,` *wts* `)`

Let *gamma* be a simple graph, and *k* a non-negative integer or vector of non-negative integers. This function returns a set *K* (possibly empty) of complete subgraphs of size *k* of *gamma*. The vertices may have weights, which should be non-zero integers if *k* is an integer and non-zero *d*-vectors of non-negative integers if *k* is a *d*-vector, and in these cases, a complete subgraph of **size** *k* means a complete subgraph whose vertex-weights sum to *k*. The exact nature of the set *K* depends on the values of the parameters supplied to this function. A complete subgraph is represented by its vertex-set.

The optional parameter *alls* controls how many complete subgraphs are returned. The valid values for *alls* are 0, 1 (the default), and 2.

Warning: Using the default value of 1 for *alls* (see below) means that more than one element may be returned for some *gamma*`.group` orbit(s) of the required complete subgraphs. To obtain just one element from each *gamma*`.group` orbit of the required complete subgraphs, you must give the value 2 to the parameter *alls*.

If *alls*=0 (or `false` for backward compatibility) then *K* will contain at most one element. If *maxi*=`false` then *K* will contain one element if and only if *gamma* contains a complete subgraph of size *k*. If *maxi*=`true` then *K* will contain one element if and only if *gamma* contains a maximal complete subgraph of size *k*, in which case *K* will contain (the vertex-set of) such a maximal complete subgraph. (A **maximal complete subgraph** of *gamma* is a complete subgraph of *gamma* which is not properly contained in another complete subgraph of *gamma*.)

If *alls*=1 (or `true` for backward compatibility) and *maxi*=`false`, then *K* will contain (perhaps properly) a set of *gamma*`.group` orbit-representatives of the size *k* complete subgraphs of *gamma*. If *alls*=1 (the default) and *maxi*=`true`, then *K* will contain (perhaps properly) a set of *gamma*`.group` orbit-representatives of the size *k* **maximal** complete subgraphs of *gamma*.

If *alls*=2 and *maxi*=`false`, then *K* will be a set of *gamma*`.group` orbit-representatives of the size *k* complete subgraphs of *gamma*. If *alls*=2 and *maxi*=`true` then *K* will be a set of *gamma*`.group` orbit-representatives of the size *k* **maximal** complete subgraphs of *gamma*. This option can be more costly than when *alls*=1.

The optional parameter *maxi* controls whether only maximal complete subgraphs of size k are returned. The default is `false`, which means that non-maximal as well as maximal complete subgraphs of size k are returned. If *maxi*=`true` then only maximal complete subgraphs of size k are returned. (Previous to version 4.1 of GRAPE, *maxi*=`true` meant that it was assumed (but not checked) that all complete subgraphs of size k were maximal.)

The optional boolean parameter *col* is used to determine whether or not partial proper vertex-colouring is used to cut down the search tree. The default is `true`, which says to use this partial colouring. For backward compatibility, *col* a rational number means the same as *col*=`true`.

The optional parameter *wts* should be a list of vertex-weights; the list should be of length *gamma*`.order`, with the i-th element being the weight of vertex i. The weights must be all positive integers if k is an integer, and all non-zero d-vectors of non-negative integers if k is a d-vector. The default is that all weights are equal to 1. (Recall that a complete subgraph of **size** k means a complete subgraph whose vertex-weights sum to k.)

If *wts* is a list of integers, then this list must be *gamma*`.group` invariant, where the action permutes the list positions in the natural way.

If *wts* is a list of d-vectors then we assume that *gamma*`.group` acts on the set of all integer d-vectors by permuting vector positions, such that, for all v in `[1..`*gamma*`.order]` and all g in *gamma*`.group`, we have $wts[v^g] = wts[v]^g$ (where the first action is `OnPoints` and for the second action, if $i^g = j$ then $(wts[v]^g)[j] = wts[v][i]$), and that we also have $k^g = k$. These assumptions are **not** checked by the function, and the use of vector-weights is primarily for advanced users of GRAPE.

An alternative name for this function is `CliquesOfGivenSize` .

See also 7.2.1.

```
gap> gamma:=JohnsonGraph(6,2);
rec( isGraph := true, order := 15,
  group := Group([ ( 1, 6,10,13,15, 5)( 2, 7,11,14, 4, 9)( 3, 8,12),
      ( 2, 6)( 3, 7)( 4, 8)( 5, 9) ]),
  schreierVector := [ -1, 2, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1 ],
  adjacencies := [ [ 2, 3, 4, 5, 6, 7, 8, 9 ] ], representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 1, 6 ], [ 2, 3 ],
      [ 2, 4 ], [ 2, 5 ], [ 2, 6 ], [ 3, 4 ], [ 3, 5 ], [ 3, 6 ], [ 4, 5 ],
      [ 4, 6 ], [ 5, 6 ] ], isSimple := true )
gap> CompleteSubgraphsOfGivenSize(gamma,4);
[ [ 1, 2, 3, 4 ] ]
gap> CompleteSubgraphsOfGivenSize(gamma,4,1,true);
[  ]
gap> CompleteSubgraphsOfGivenSize(gamma,5,2,true);
[ [ 1, 2, 3, 4, 5 ] ]
gap> delta:=NewGroupGraph(Group(()),gamma);;
gap> CompleteSubgraphsOfGivenSize(delta,5,2,true);
[ [ 1, 2, 3, 4, 5 ], [ 1, 6, 7, 8, 9 ], [ 2, 6, 10, 11, 12 ],
  [ 3, 7, 10, 13, 14 ], [ 4, 8, 11, 13, 15 ], [ 5, 9, 12, 14, 15 ] ]
gap> CompleteSubgraphsOfGivenSize(delta,5,0);
[ [ 1, 2, 3, 4, 5 ] ]
gap> CompleteSubgraphsOfGivenSize(delta,5,1,false,true,
>       [1,2,3,4,5,6,7,8,7,6,5,4,3,2,1]);
[ [ 1, 4 ], [ 2, 3 ], [ 3, 14 ], [ 4, 15 ], [ 5 ], [ 11 ], [ 12, 15 ],
  [ 13, 14 ] ]
```

8 Automorphism groups and isomorphism testing for graphs

GRAPE includes B.D. McKay's *nauty* (Version 2.2, final patched version) package for calculating automorphism groups of graphs and for testing graph isomorphism (see [Nau90,MP14]). As described in Section 1.1, a user may instead use their own copy of dreadnaut or dreadnautB from a later version of *nauty*, or may use T. Juntilla's and P. Kaski's *bliss* package [JK07] instead of *nauty*. Many functions described in this chapter make use of *nauty* or *bliss*.

8.1 Graphs with colour-classes

For each of the functions described in this chapter, each graph parameter may be replaced by a graph with colour-classes, which is a record having (at least) the components `graph` (which should be a graph in GRAPE format), and `colourClasses`, which should be an ordered partition of the vertices of the graph, and so define **colour-classes** for the vertices. This ordered partition should be given as a list of (pairwise-disjoint non-empty) sets partitioning the vertex-set. When these functions are called with graphs with colour-classes, then it is understood that an **automorphism** of a graph with colour-classes is an automorphism of the graph which additionally preserves the list of colour-classes (classwise), and an **isomorphism** from one graph with colour-classes to a second is a graph isomorphism from the first graph to the second which additionally maps the first list of colour-classes to the second (classwise). The record for a graph with colour-classes may also optionally contain the additional components `autGroup` and/or `canonicalLabelling`, and these are handled in an analogous way to those for a graph (such as when using the parameter *firstunbindcanon*). Note that we do not require that adjacent vertices be in different colour-classes.

8.2 AutGroupGraph

1 ▶ `AutGroupGraph(` *gamma* `)`
▶ `AutGroupGraph(` *gamma*`,` *colourclasses* `)`

The first version of this function returns the automorphism group of the graph (or graph with colour-classes) *gamma*, using *nauty* or *bliss* (this can also be accomplished by typing `AutomorphismGroup(`*gamma*`)`). The **automorphism group** Aut(*gamma*) of a graph *gamma* is the group consisting of the permutations of the vertices of *gamma* which preserve the edge-set of *gamma*. The **automorphism group** of a graph with colour-classes is the subgroup of the automorphism group of the graph which preserves the colour-classes (classwise).

The second version of this function is maintained only for backward compatibility. For this version *gamma* must be a graph, *colourclasses* is an ordered partition of the vertices of *gamma*, and the subgroup of Aut(*gamma*) preserving this ordered partition is returned. The ordered partition should be given as a list of (pairwise-disjoint non-empty) sets partitioning the vertices of *gamma*, although for backward compatibility and only in this situation, the last set in the ordered partition need not be included explicitly.

```
gap> gamma := JohnsonGraph(4,2);
rec( adjacencies := [ [ 2, 3, 4, 5 ] ],
  group := Group([ (1,4,6,3)(2,5), (2,4)(3,5) ]), isGraph := true,
  isSimple := true,
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ],
  order := 6, representatives := [ 1 ],
  schreierVector := [ -1, 2, 1, 1, 1, 1 ] )
gap> Size(AutGroupGraph(gamma));
48
gap> AutGroupGraph( rec(graph:=gamma,colourClasses:=[[1,2,3],[4,5,6]]) );
Group([ (2,3)(4,5), (1,2)(5,6) ])
gap> Size(AutomorphismGroup( rec(graph:=gamma,colourClasses:=[[1,6],[2,3,4,5]]) ));
16
```

8.3 GraphIsomorphism

1 ▶ `GraphIsomorphism(` *gamma1*, *gamma2* `)`
▶ `GraphIsomorphism(` *gamma1*, *gamma2*, *firstunbindcanon* `)`

Let *gamma1* and *gamma2* both be graphs or both be graphs with colour-classes. Then this function makes use of *nauty* or *bliss* to (try to) determine an isomorphism from *gamma1* to *gamma2*. If *gamma1* and *gamma2* are isomorphic, then this function returns an isomorphism from *gamma1* to *gamma2*. This isomorphism will be a permutation of the vertices of *gamma1* which maps the edge-set of *gamma1* onto that of *gamma2*, and if *gamma1* and *gamma2* are graphs with colour-classes, this isomorphism will also map the colour-class list of *gamma1* to that of *gamma2* (classwise). If *gamma1* and *gamma2* are not isomorphic then this function returns `fail`.

The optional boolean parameter *firstunbindcanon* determines whether or not the `canonicalLabelling` components of both *gamma1* and *gamma2* are first unbound before proceeding. If *firstunbindcanon* is `true` (the default, safe and possibly slower option) then these components are first unbound. If *firstunbindcanon* is `false`, then any existing `canonicalLabelling` components are used. However, since canonical labellings can depend on whether *nauty* or *bliss* is used, the version of *nauty* or *bliss* used, the version of GRAPE, parameter settings of *nauty* or *bliss*, and possibly even the compiler and computer used, you must be sure that if *firstunbindcanon*=`false` then the `canonicalLabelling` component(s) which may already exist for *gamma1* or *gamma2* were created in exactly the same environment in which you are presently computing.

Please also note that a canonical labelling for a GRAPE graph is the inverse of what a canononical labelling for a graph is usually defined as (such as in *bliss*), in that in GRAPE, the image of a graph under the **inverse** of its canonical labelling is the calculated canonical version of that graph.

See also 8.4.1.

```
gap> gamma := JohnsonGraph(5,3);
rec( adjacencies := [ [ 2, 3, 4, 5, 7, 8 ] ],
  group := Group([ (1,7,10,6,3)(2,8,4,9,5), (4,7)(5,8)(6,9) ]),
  isGraph := true, isSimple := true,
  names := [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 2, 5 ], [ 1, 3, 4 ], [ 1, 3, 5 ],
      [ 1, 4, 5 ], [ 2, 3, 4 ], [ 2, 3, 5 ], [ 2, 4, 5 ], [ 3, 4, 5 ] ],
  order := 10, representatives := [ 1 ],
  schreierVector := [ -1, 1, 1, 2, 1, 1, 1, 2, 1, 1 ] )
gap> delta := JohnsonGraph(5,2);
rec( adjacencies := [ [ 2, 3, 4, 5, 6, 7 ] ],
  group := Group([ (1,5,8,10,4)(2,6,9,3,7), (2,5)(3,6)(4,7) ]),
  isGraph := true, isSimple := true,
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ], [ 2, 4 ],
      [ 2, 5 ], [ 3, 4 ], [ 3, 5 ], [ 4, 5 ] ], order := 10,
```

```
  representatives := [ 1 ], schreierVector := [ -1, 2, 2, 1, 1, 1, 2, 1, 1, 1
     ] )
gap> GraphIsomorphism( gamma, delta );
(3,5,6,8,7,4)
gap> GraphIsomorphism(
>       rec(graph:=gamma, colourClasses:=[[7],[1,2,3,4,5,6,8,9,10]]),
>       rec(graph:=delta, colourClasses:=[[10],[1..9]]) );
(1,3)(2,6,5)(4,8)(7,10,9)
gap> GraphIsomorphism(
>       rec(graph:=gamma, colourClasses:=[[1],[6],[2,3,4,5,7,8,9,10]]),
>       rec(graph:=delta, colourClasses:=[[1],[6],[2,3,4,5,7,8,9,10]]) );
fail
```

8.4 IsIsomorphicGraph

1 ▶ `IsIsomorphicGraph( gamma1, gamma2 )`
▶ `IsIsomorphicGraph( gamma1, gamma2, firstunbindcanon )`

Let *gamma1* and *gamma2* both be graphs or both be graphs with colour-classes. Then this boolean function makes use of the *nauty* or *bliss* package to test whether *gamma1* and *gamma2* are isomorphic (as graphs or as graphs with colour-classes, respectively). The value `true` is returned if and only if the graphs (or graphs with colour-classes) are isomorphic.

The optional boolean parameter *firstunbindcanon* determines whether or not the `canonicalLabelling` components of both *gamma1* and *gamma2* are first unbound before proceeding. If *firstunbindcanon* is `true` (the default, safe and possibly slower option) then these components are first unbound. If *firstunbindcanon* is `false`, then any existing `canonicalLabelling` components are used. However, since canonical labellings can depend on whether *nauty* or *bliss* is used, the version of *nauty* or *bliss* used, the version of GRAPE, parameter settings of *nauty* or *bliss*, and possibly even the compiler and computer used, you must be sure that if *firstunbindcanon*=`false` then the `canonicalLabelling` component(s) which may already exist for *gamma1* or *gamma2* were created in exactly the same environment in which you are presently computing.

See also 8.3.1. For pairwise isomorphism testing of three or more graphs (or graphs with colour-classes), see 8.5.1.

Please also note that a canonical labelling for a GRAPE graph is the inverse of what a canononical labelling for a graph is usually defined as (such as in *bliss*), in that in GRAPE, the image of a graph under the **inverse** of its canonical labelling is the calculated canonical version of that graph.

```
gap> gamma := JohnsonGraph(5,3);
rec( adjacencies := [ [ 2, 3, 4, 5, 7, 8 ] ],
  group := Group([ (1,7,10,6,3)(2,8,4,9,5), (4,7)(5,8)(6,9) ]),
  isGraph := true, isSimple := true,
  names := [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 2, 5 ], [ 1, 3, 4 ], [ 1, 3, 5 ],
      [ 1, 4, 5 ], [ 2, 3, 4 ], [ 2, 3, 5 ], [ 2, 4, 5 ], [ 3, 4, 5 ] ],
  order := 10, representatives := [ 1 ],
  schreierVector := [ -1, 1, 1, 2, 1, 1, 1, 2, 1, 1 ] )
gap> delta := JohnsonGraph(5,2);
rec( adjacencies := [ [ 2, 3, 4, 5, 6, 7 ] ],
  group := Group([ (1,5,8,10,4)(2,6,9,3,7), (2,5)(3,6)(4,7) ]),
  isGraph := true, isSimple := true,
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ], [ 2, 4 ],
      [ 2, 5 ], [ 3, 4 ], [ 3, 5 ], [ 4, 5 ] ], order := 10,
  representatives := [ 1 ], schreierVector := [ -1, 2, 2, 1, 1, 1, 2, 1, 1, 1
     ] )
gap> IsIsomorphicGraph( gamma, delta );
```

```
true
gap> IsIsomorphicGraph(
>       rec(graph:=gamma, colourClasses:=[[7],[1,2,3,4,5,6,8,9,10]]),
>       rec(graph:=delta, colourClasses:=[[10],[1..9]]) );
true
gap> IsIsomorphicGraph(
>       rec(graph:=gamma, colourClasses:=[[1],[6],[2,3,4,5,7,8,9,10]]),
>       rec(graph:=delta, colourClasses:=[[1],[6],[2,3,4,5,7,8,9,10]]) );
false
```

8.5 GraphIsomorphismClassRepresentatives

1 ▶ `GraphIsomorphismClassRepresentatives( L )`
▶ `GraphIsomorphismClassRepresentatives( L, firstunbindcanon )`

Given a list *L* of graphs, or of graphs with colour-classes, this function uses *nauty* or *bliss* to return a list consisting of pairwise non-isomorphic elements of *L*, representing all the isomorphism classes of elements of *L*.

The optional boolean parameter *firstunbindcanon* determines whether or not the `canonicalLabelling` components of all elements of *L* are first unbound before proceeding. If *firstunbindcanon* is `true` (the default, safe and possibly slower option) then these components are first unbound. If *firstunbindcanon* is `false`, then any existing `canonicalLabelling` components of elements of *L* are used. However, since canonical labellings can depend on whether *nauty* or *bliss* is used, the version of *nauty* or *bliss* used, the version of GRAPE, parameter settings of *nauty* or *bliss*, and possibly even the compiler and computer used, you must be sure that if *firstunbindcanon*=`false` then any `canonicalLabelling` component(s) which may already exist for elements of *L* were created in exactly the same environment in which you are presently computing.

It is assumed that the computing environment is constant throughout the execution of this function.

Please also note that a canonical labelling for a GRAPE graph is the inverse of what a canononical labelling for a graph is usually defined as (such as in *bliss*), in that in GRAPE, the image of a graph under the **inverse** of its canonical labelling is the calculated canonical version of that graph.

```
gap> A:=JohnsonGraph(5,3);
rec( adjacencies := [ [ 2, 3, 4, 5, 7, 8 ] ],
  group := Group([ (1,7,10,6,3)(2,8,4,9,5), (4,7)(5,8)(6,9) ]),
  isGraph := true, isSimple := true,
  names := [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 2, 5 ], [ 1, 3, 4 ], [ 1, 3, 5 ],
      [ 1, 4, 5 ], [ 2, 3, 4 ], [ 2, 3, 5 ], [ 2, 4, 5 ], [ 3, 4, 5 ] ],
  order := 10, representatives := [ 1 ],
  schreierVector := [ -1, 1, 1, 2, 1, 1, 1, 2, 1, 1 ] )
gap> B:=JohnsonGraph(5,2);
rec( adjacencies := [ [ 2, 3, 4, 5, 6, 7 ] ],
  group := Group([ (1,5,8,10,4)(2,6,9,3,7), (2,5)(3,6)(4,7) ]),
  isGraph := true, isSimple := true,
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ], [ 2, 4 ],
      [ 2, 5 ], [ 3, 4 ], [ 3, 5 ], [ 4, 5 ] ], order := 10,
  representatives := [ 1 ], schreierVector := [ -1, 2, 2, 1, 1, 1, 2, 1, 1, 1
     ] )
gap> R:=GraphIsomorphismClassRepresentatives([A,B,ComplementGraph(A)]);;
gap> Length(R);
2
gap> List(R,VertexDegrees);
[ [ 6 ], [ 3 ] ]
gap> R:=GraphIsomorphismClassRepresentatives(
```

```
>    [ rec(graph:=gamma, colourClasses:=[[1],[6],[2,3,4,5,7,8,9,10]]),
>      rec(graph:=delta, colourClasses:=[[1],[6],[2,3,4,5,7,8,9,10]]),
>      rec(graph:=ComplementGraph(gamma), colourClasses:=[[1],[6],[2,3,4,5,7,8,9,10]]) ] );;
gap> Length(R);
3
```

9 Partial Linear Spaces

Let s and t be positive integers. A **partial linear space** (P,L), with **parameters** (s,t) consists of a set P of **points**, together with a set L of $(s+1)$-subsets of P called **lines**, such that every point is in exactly $t+1$ lines, and every pair of distinct points is contained in at most one line. The **point graph** of a partial linear space S having point-set P is the graph with vertex-set P and having $[p,q]$ an edge if and only if $p \neq q$ and p,q are in a common line of S. Two partial linear spaces (P,L) and (P',L') (with parameters (s,t)) are said to be **isomorphic** if there is a bijection $P \to P'$ which induces a bijection $L \to L'$. An **automorphism** of a partial linear space is an isomorphism onto itself. The set of all automorphisms of a partial linear space S forms a group, called the **automorphism group** of S.

GRAPE contains a function `PartialLinearSpaces` to determine and classify partial linear spaces with given point graph and parameters. In this chapter we describe this function, and also give a research application of this function.

9.1 PartialLinearSpaces

1 ▶ `PartialLinearSpaces(` *ptgraph*`,` *s*`,` *t* `)`
▶ `PartialLinearSpaces(` *ptgraph*`,` *s*`,` *t*`,` *nspaces* `)`
▶ `PartialLinearSpaces(` *ptgraph*`,` *s*`,` *t*`,` *nspaces*`,` *printlevel* `)`
▶ `PartialLinearSpaces(` *ptgraph*`,` *s*`,` *t*`,` *nspaces*`,` *printlevel*`,` *cliques* `)`

This function classifies the partial linear spaces with given point graph *ptgraph* and parameters (*s*,*t*). It makes use (indirectly) of *nauty* [Nau90,MP14] or *bliss* [JK07].

The function `PartialLinearSpaces` returns a list of representatives of distinct isomorphism classes of partial linear spaces with (simple) point graph *ptgraph*, and parameters (*s*,*t*). The default is that representatives for all isomorphism classes are returned.

The integer argument *nspaces* is optional, and has default value -1, which means that representatives for all isomorphism classes are returned. If *nspaces* is non-negative then exactly *nspaces* representatives are returned if there are at least *nspaces* isomorphism classes, otherwise representatives for all isomorphism classes are returned.

In the output of this function, a partial linear space S is given by its incidence graph *delta*. The point-vertices of *delta* are 1,...,*ptgraph*`.order`, with the name of point-vertex *i* being the name of vertex *i* of *ptgraph*. A line-vertex of *delta* is named by a list (not necessarily ordered) of the point-vertex names for the points on that line. We warn that this is a **different** naming convention to versions of GRAPE before 4.1. The group *delta*`.group` associated with the incidence graph *delta* is the automorphism group of S acting on point-vertices and line-vertices, and preserving both sets.

If *printlevel* is bound then it controls the print-level (default 0). Permitted values for *printlevel* are 0,1,2.

If *cliques* is bound then it is assumed to be a list (without repeats) of the $(s+1)$-cliques of *ptgraph*. If known, this can help the function to run faster.

```
gap> K7:=CompleteGraph(SymmetricGroup(7));;
gap> P:=PartialLinearSpaces(K7,2,2);
[ rec( isGraph := true, order := 14,
      group := Group([ ( 1, 2)( 5, 6)( 9,11)(10,12),
          ( 1, 2, 3)( 5, 6, 7)( 9,11,13)(10,12,14),
          ( 1, 2, 3)( 4, 7, 6)( 9,12,14)(10,11,13),
          ( 1, 4, 7, 6, 2, 5, 3)( 8, 9,13,10,11,12,14) ]),
```

```
      schreierVector := [ -1, 1, 2, 4, 4, 1, 3, -2, 4, 1, 1, 3, 4, 2 ],
      adjacencies := [ [ 8, 9, 10 ], [ 1, 2, 3 ] ],
      representatives := [ 1, 8 ],
      names := [ 1, 2, 3, 4, 5, 6, 7, [ 1, 2, 3 ], [ 1, 4, 5 ], [ 1, 6, 7 ],
          [ 2, 4, 6 ], [ 2, 5, 7 ], [ 3, 4, 7 ], [ 3, 5, 6 ] ],
      isSimple := true ) ]
gap> Size(P[1].group);
168
gap> T:=ComplementGraph(JohnsonGraph(10,2));;
gap> P:=PartialLinearSpaces(T,4,6);;
gap> List(P,x->Size(x.group));
[ 216, 1512 ]
```

9.2 A research application of PartialLinearSpaces

We now provide an extended example of the use of GRAPE which illustrates a research application of the `PartialLinearSpaces` function.

First we give a definition. Let s and t be positive integers. A **partial geometry** is a partial linear space with parameters (s,t) for which there is an additional constant constant $\alpha > 0$, such that, for every line l and every point p not on l, there are exactly α lines through p meeting l in some point.

Our example shows that the Haemers partial geometry [Hae81] is uniquely determined (up to isomorphism) by its point graph, as is the dual of the Haemers geometry (where the role of points and lines are interchanged), and that each of these geoemetries has automorphism group isomorphic to A_7.

We first construct and study the Hoffman-Singleton graph, using the construction of Peter Cameron contained in [Cam99]. We then construct the point graph of the Haemers partial geometry [Hae81] (this partial geometry has $(s,t) = (4,17)$ and $\alpha = 2$). The vertices of this point graph are the edges of the Hoffman-Singleton graph, and two such vertices are adjacent in the point graph precisely when they are at distance 2 in the edge-graph of the Hoffman-Singleton graph (see [Hae81]). We then construct and classify (up to isomorphism) all partial linear spaces with parameters $(4,17)$ having point graph isomorphic to that of the Haemers partial geometry. We find that the Haemers partial geometry is the only possibility. It follows from basic theory of partial geometries that the Haemers partial geometry is uniquely determined up to isomorphism (as a partial geometry) by its point graph. We also show that the dual of the Haemers partial geometry is also uniquely determined by its point graph. Thus far, the only proof of these results is by GRAPE. Our example also shows that the Haemers partial geometry and its dual each has automorphism group isomorphic to A_7.

The total runtime (not including calls of *nauty*) was about 300 CPU-seconds on a Pentium II running at 350 MHz.

```
gap> LoadPackage("grape");
true
gap>
gap> OnSetsRecursive:=function(x,g)
> if not IsList(x) then
>   return x^g;
> else
>   return Set(List(x, y->OnSetsRecursive(y,g)));
> fi;
> end;;
gap>
gap> HofSingAdjacency := function(x,y)
> #
> # This boolean function returns  true  iff  x  and  y  are
> # adjacent in the Hoffman-Singleton graph, in Peter Cameron's
```

```
> # construction.
> #
> if Size(x)=3 then                  # x is a 3-set
>    if Size(y)=3 then               # y is a 3-set
>       return Intersection(x,y)=[]; # join iff disjoint
>    else                            # y is a projective plane
>       return x in y;               # join iff x is a line of y
>    fi;
> else                               # x is a projective plane
>    if Size(y)=3 then               # y is a 3-set
>       return y in x;               # join iff y is a line of x
>    else                            # y is a projective plane
>       return false;                # don't join
>    fi;
> fi;
> end;;
gap>
gap> projectiveplane:=
>    Set([[1,2,4],[2,3,5],[3,4,6],[4,5,7],[1,5,6],[2,6,7],[1,3,7]]);;
gap>
gap> HofSingGraph:=Graph(AlternatingGroup(7),
>                    [[1,2,3], projectiveplane], OnSetsRecursive,
>                    HofSingAdjacency);;
gap> GlobalParameters(HofSingGraph);
[ [ 0, 0, 7 ], [ 1, 0, 6 ], [ 1, 6, 0 ] ]
gap> autgrp := AutGroupGraph(HofSingGraph);;
gap> Size(autgrp);
252000
gap> HofSingGraph := NewGroupGraph(autgrp,HofSingGraph);;
gap> pointgraph:=DistanceGraph( EdgeGraph(HofSingGraph), 2);;
gap> GlobalParameters(pointgraph);
[ [ 0, 0, 72 ], [ 1, 20, 51 ], [ 36, 36, 0 ] ]
gap> spaces:=PartialLinearSpaces(pointgraph,4,17);;
gap> Length(spaces);
1
gap> haemers:=spaces[1];;
gap> DisplayCompositionSeries(haemers.group);
G (3 gens, size 2520)
  A(7)
1 (0 gens, size 1)
gap> linegraph:=PointGraph(haemers, Adjacency(haemers,1)[1]);;
gap> spaces:=PartialLinearSpaces(linegraph,17,4);;
gap> Length(spaces);
1
gap> dualhaemers:=spaces[1];;
gap> DisplayCompositionSeries(dualhaemers.group);
G (4 gens, size 2520)
  A(7)
1 (0 gens, size 1)
```

Bibliography

[BCN89] A. E. Brouwer, A. M. Cohen, and A. Neumaier. *Distance-Regular Graphs*. Springer, Berlin, Heidelberg and New York, 1989.

[Cam99] Peter J. Cameron. *Permutation Groups*. Cambridge University Press, 1999. book's web-page:

`http://www.maths.qmul.ac.uk/~pjc/permgps/pgbook.html`.

[CSS99] Hans Cuypers, Leonard H. Soicher, and Hans Sterk. The small mathieu groups (project). In Arjeh M. Cohen, Hans Cuypers, and Hans Sterk, editors, *Some Tapas of Computer Algebra*, volume 4 of *Algorithms and Computation in Mathematics*, pages 323–337. Springer, Berlin, Heidelberg and New York, 1999.

[Hae81] Willem Haemers. A new partial geometry constructed from the hoffman-singleton graph. In P. J. Cameron, J. W. P. Hirschfeld, and D. R. Hughes, editors, *Finite Geometries and Designs: Proceedings of the Second Isle of Thorns Conference 1980*, volume 49 of *London Mathematical Society Lecture Note Series*, pages 119–127. Cambridge University Press, 1981.

[HL99] Alexander Hulpke and Steve Linton. Construction of Co3. An example of the use of an integrated system for Computational Group Theory. In C. M. Campbell, E. F. Robertson, N. Ruskuc, and G. C. Smith, editors, *Groups St Andrews 1997 in Bath*, volume 260/261 of *London Mathematical Society Lecture Note Series*, pages 394–409. Cambridge University Press, 1999.

[JK07] Tommi Juntilla and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In David Applegate et al., editor, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithmics and Combinatorics*, pages 135–149. SIAM, 2007. bliss homepage:

`http://www.tcs.hut.fi/Software/bliss/`.

[McK90] Brendan D. McKay. *nauty user's guide (version 1.5), Technical report TR-CS-90-02*. Australian National University, Computer Science Department, 1990. nauty homepage:

`http://cs.anu.edu.au/people/bdm/nauty/`.

[MP14] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, ii. *J. Symbolic Comput.*, 60:94–112, 2014.

[PS97] Cheryl E. Praeger and Leonard H. Soicher. *Low rank representations and graphs for sporadic groups*, volume 8 of *Australian Mathematical Society Lecture Series*. Cambridge University Press, 1997.

[Soi93] Leonard H. Soicher. GRAPE: a system for computing with graphs and groups. In Larry Finkelstein and William M. Kantor, editors, *Groups and Computation*, volume 11 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 287–291. American Mathematical Society, 1993. GRAPE homepage:

`http://www.maths.qmul.ac.uk/~leonard/grape/`.

[Soi04] Leonard H. Soicher. Computing with graphs and groups. In Lowell W. Beineke and Robin J. Wilson, editors, *Topics in Algebraic Graph Theory*, volume 102 of *Encyclopedia of Mathematics and its Applications*, pages 250–266. Cambridge University Press, 2004.

[Soi06] Leonard H. Soicher. Is there a McLaughlin geometry? *J. Algebra*, 300:248–255, 2006.

Index

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