Digraphs

Methods for digraphs

Version 0.4.2

Jan De Beule
Julius Jonušas
James D. Mitchell
Michael Torpey
Wilf Wilson

Jan De Beule Email: jdebeule@cage.ugent.be
Homepage: http://homepages.vub.ac.be/~jdbeule/

Julius Jonušas Email: jj252@st-and.ac.uk
Homepage: http://www-circa.mcs.st-and.ac.uk/~julius

James D. Mitchell Email: jdm3@st-and.ac.uk
Homepage: http://tinyurl.com/jdmitchell

Michael Torpey Email: mct25@st-and.ac.uk
Homepage: http://www-circa.mcs.st-and.ac.uk/~mct25

Wilf Wilson Email: waw7@st-and.ac.uk
Homepage: http://www-circa.mcs.st-and.ac.uk/~waw7

Abstract

The Digraphs package is a GAP package containing methods for graphs, digraphs, and multidigraphs.

Copyright

Acknowledgements

We would like to thank Chris Jefferson for his help in including the bliss tool in the package. This package's methods for computing digraph homomorphisms are based on work by Max Neunhöffer, and independently Artur Schäfer.

Contents

Chapter 1

The Digraphs package

1.1 Introduction

This is the manual for the Digraphs package version 0.4.2. This package was developed at the University of St Andrews by:

- Jan De Beule
- Julius Jonusas
- James D. Mitchell
- Michael C. Torpey
- Wilf A. Wilson

The Digraphs package contains a variety of methods for efficiently creating and storing digraphs and computing information about them. Full explanations of all the functions contained in the package are provided below.

If the Grape package is available, it will be loaded automatically. Digraphs created with the Digraphs package can be converted to Grape graphs with `Graph` (3.2.3), and conversely Grape graphs can be converted to Digraphs objects with `Digraph` (3.1.5). Grape is not required for Digraphs to run.

The bliss tool [JK07] is included in this package. It is an open-source tool for computing automorphism groups and canonical forms of graphs, written by Tommi Junttila and Petteri Kaski. Several of the methods in the Digraphs package rely on bliss.

For the purposes of this package and its documentation, the following definitions apply:

A *digraph* $E = (E^0, E^1, r, s)$, also known as a *directed graph*, consists of a set of vertices E^0 and a set of edges E^1 together with functions $s, r : E^1 \to E^0$, called the *source* and *range*, respectively. The source and range of an edge is respectively the values of s, r at that edge. An edge is called a *loop* if its source and range are the same. A digraph is called a *multidigraph* if there exist two or more edges with the same source and the same range.

A *directed path* on a digraph is a sequence of alternating vertices and edges $(v_1, e_1, v_2, e_2, ..., e_{n-1}, v_n)$ such that each edge e_i has source v_i and range v_{i+1}, and no vertex is repeated. A *directed walk* is defined similarly, but vertices may be repeated. A *cycle* is defined similarly to a directed path, except that $v_1 = v_n$. The *length* of a directed path or cycle $(v_1, e_1, v_2, e_2, ..., e_{n-1}, v_n)$ is equal to $n-1$, the number of edges it contains.

Chapter 2

Installing Digraphs

2.1 For those in a hurry

In this section we give a brief description of how to start using Digraphs.

It is assumed that you have a working copy of GAP with version number 4.8.2 or higher. The most up-to-date version of GAP and instructions on how to install it can be obtained from the main GAP webpage `http://www.gap-system.org`.

The following is a summary of the steps that should lead to a successful installation of Digraphs:

- ensure that the IO package version 4.4.4 or higher is available. IO must be compiled before Digraphs can be loaded.
- THIS STEP IS OPTIONAL: certain functions in Digraphs require the Grape package to be available; see Section 2.2.1 for full details. To use these functions make sure that the Grape package version 4.5 or higher is available. If Grape is not available, then Digraphs can be used as normal with the exception that the functions listed in Subsection 2.2.1 will not work.
- download the package archive `digraphs-0.4.2.tar.gz` from the Digraph package webpage.
- unzip and untar the file, this should create a directory called `digraphs-0.4.2`.
- locate the `pkg` directory of your GAP directory, which contains the directories `lib`, `doc` and so on. Move the directory `digraphs-0.4.2` into the `pkg` directory.
- it is necessary to compile the Digraphs package. Inside the `pkg/digraphs-0.4.2` directory, type

  ```
  ./configure
  make
  ```

  Further information about this step can be found in Section 2.3.
- start GAP in the usual way (i.e. type gap at the command line).
- type `LoadPackage("digraphs");`

If you want to check that the package is working correctly, you should run some of the tests described in Section 2.5.

2.2 Optional package dependencies

The Digraphs package is written in GAP and C code and requires the IO package. The IO package is used to read and write transformations, partial permutations, and bipartitions to a file.

2.2.1 The Grape package

The Grape package must be available for the following operations to be available:

- `Graph` (3.2.3) with a digraph argument
- `AsGraph` (3.2.4) with a digraph argument
- `Digraph` (3.1.5) with a Grape graph argument

If Grape is not available, then Digraphs can be used as normal with the exception that the functions above will not work.

2.3 Compiling the kernel module

The Digraphs package has a GAP kernel component in C which should be compiled. This component contains certain low-level functions required by Digraphs.

It is not possible to use the Digraphs package without compiling it.

To compile the kernel component inside the `pkg/digraphs-0.4.2` directory, type

```
./configure
make
```

If you installed the package in another 'pkg' directory than the standard 'pkg' directory in your GAP installation, then you have to do two things. Firstly during compilation you have to use the option '–with-gaproot=PATH' of the 'configure' script where 'PATH' is a path to the main GAP root directory (if not given the default '../..' is assumed).

If you installed GAP on several architectures, you must execute the configure/make step for each of the architectures. You can either do this immediately after configuring and compiling GAP itself on this architecture, or alternatively (when using version 4.5 of GAP or newer) set the environment variable 'CONFIGNAME' to the name of the configuration you used when compiling GAP before running './configure'. Note however that your compiler choice and flags (environment variables 'CC' and 'CFLAGS') need to be chosen to match the setup of the original GAP compilation. For example you have to specify 32-bit or 64-bit mode correctly!

2.4 Rebuilding the documentation

The Digraphs package comes complete with pdf, html, and text versions of the documentation. However, you might find it necessary, at some point, to rebuild the documentation. To rebuild the documentation use the `DigraphsMakeDoc` (2.4.1).

2.4.1 DigraphsMakeDoc

▷ `DigraphsMakeDoc()` (function)

Returns: Nothing

This function should be called with no argument to compile the Digraphs documentation.

2.5 Testing your installation

In this section we describe how to test that Digraphs is working as intended. To test that Digraphs is installed correctly use `DigraphsTestInstall` (2.5.1) or for more extensive tests use `DigraphsTestStandard` (2.5.2).

If something goes wrong, then please review the instructions in Section 2.1 and ensure that Digraphs has been properly installed. If you continue having problems, please use the issue tracker to report the issues you are having.

2.5.1 DigraphsTestInstall

▷ `DigraphsTestInstall()` (function)

Returns: `true` or `false`.

This function should be called with no argument to test your installation of Digraphs is working correctly. These tests should take no more than a fraction of a second to complete. To test more comprehensively that Digraphs is working correctly, use `DigraphsTestStandard` (2.5.2).

2.5.2 DigraphsTestStandard

▷ `DigraphsTestStandard()` (function)

Returns: `true` or `false`.

This function should be called to test all the methods included in Digraphs. These tests should take only a few seconds to complete.

To quickly test that Digraphs is installed correctly use `DigraphsTestInstall` (2.5.1). For a more thorough test, use `DigraphsTestStandard`.

Chapter 3

Creating digraphs

In this chapter we describe how to create digraphs.

3.1 Creating digraphs

3.1.1 IsDigraph

▷ `IsDigraph` (Category)

Every digraph in Digraphs belongs to the category `IsDigraph`. Basic attributes and operations for digraphs are: `DigraphVertices` (5.1.1), `DigraphRange` (5.2.4), `DigraphSource` (5.2.4), `OutNeighbours` (5.2.5), and `DigraphEdges` (5.1.3).

3.1.2 IsCayleyDigraph

▷ `IsCayleyDigraph` (Category)

`IsCayleyDigraph` is a subcategory of `IsDigraph`. Digraphs that are Cayley digraphs of a group and that are constructed by the operation `CayleyDigraph` (3.5.6) are constructed in this category.

3.1.3 IsDigraphWithAdjacencyFunction

▷ `IsDigraphWithAdjacencyFunction` (Category)

`IsCayleyDigraph` is a subcategory of `IsDigraph`. Digraphs that are *created* using an adjacency function are constructed in this category

3.1.4 DigraphType

▷ `DigraphType` (global variable)
▷ `DigraphFamily` (global variable)

The type of all digraphs is `DigraphType`. The family of all digraphs is `DigraphFamily`.

3.1.5 Digraph

▷ `Digraph(obj[, source, range])` (operation)
▷ `Digraph(list, func)` (operation)
▷ `Digraph(G, list, act, adj)` (operation)

Returns: A digraph.

for a list (i.e. an adjacency list)

if *obj* is a list of lists of positive integers in the range from `1` to `Length(obj)`, then this function returns the digraph with vertices $E^0 =$ `[1 .. Length(obj)]`, and edges corresponding to the entries of *obj*.

More precisely, there is an edge from vertex `i` to `j` if and only if `j` is in *obj*`[i]`; the source of this edge is `i` and the range is `j`. If `j` occurs in *obj*`[i]` with multiplicity `k`, then there are `k` edges from `i` to `j`.

for three lists

if *obj* is a duplicate-free list, and *source* and *range* are lists of equal length consisting of positive integers in the list `[1 .. Length(obj)]`, then this function returns a digraph with vertices $E^0 =$ `[1 .. Length(obj)]`, and `Length(source)` edges. For each `i` in `[1 .. Length(source)]` there exists an edge with source vertex `source[i]` and range vertex `range[i]`. See `DigraphSource` (5.2.4) and `DigraphRange` (5.2.4).

The vertices of the digraph will be labelled by the elements of *obj*.

for an integer, and two lists

if *obj* is an integer, and *source* and *range* are lists of equal length consisting of positive integers in the list `[1 .. obj]`, then this function returns a digraph with vertices $E^0 =$ `[1 .. obj]`, and `Length(source)` edges. For each `i` in `[1 .. Length(source)]` there exists an edge with source vertex `source[i]` and range vertex `range[i]`. See `DigraphSource` (5.2.4) and `DigraphRange` (5.2.4).

for a list and a function

if *list* is a list and *func* is a function taking 2 arguments that are elements of *list*, and *func* returns `true` or `false`, then this operation creates a digraph with vertices `[1 .. Length(list)]` and an edge from vertex `i` to vertex `j` if and only if `func(list[i], list[j])` returns `true`.

for a group, a list, and two functions

The arguments will be *G, list, act, adj*.

Let *G* be a group acting on the objects in *list* via the action *act*, and let *adj* be a function taking two objects from *list* as arguments and returning `true` or `false`. The function *adj* will describe the adjacency between objects from *list*, which is invariant under the action of *G*. This variant of the constructor returns a digraph with vertices the objects of *list* and directed edges `[x, y]` when `f(x, y)` is `true`.

The action of the group *G* on the objects in *list* is stored in the attribute `DigraphGroup` (7.2.3), and is used to speed up operations like `DigraphDiameter` (5.3.1).

for a Grape package graph

if *obj* is a Grape package graph (i.e. a record for which the function `IsGraph` returns `true`), then this function returns a digraph isomorphic to *obj*.

for a binary relation

if *obj* is a binary relation on the points [1 .. n] for some posititve integer n, then this function returns the digraph defined by *obj*. Specifically, this function returns a digraph which has n vertices, and which has an edge with source i and range j if and only if [i,j] is a pair in the binary relation *obj*.

Example
```
gap> gr := Digraph([[2, 5, 8, 10], [2, 3, 4, 2, 5, 6, 8, 9, 10],
> [1], [3, 5, 7, 8, 10], [2, 5, 7], [3, 6, 7, 9, 10], [1, 4],
> [1, 5, 9], [1, 2, 7, 8], [3, 5]]);
<multidigraph with 10 vertices, 38 edges>
gap> gr := Digraph(["a", "b", "c"], ["a"], ["b"]);
<digraph with 3 vertices, 1 edge>
gap> gr := Digraph(5, [1, 2, 2, 4, 1, 1], [2, 3, 5, 5, 1, 1]);
<multidigraph with 5 vertices, 6 edges>
gap> Petersen := Graph(SymmetricGroup(5), [[1, 2]], OnSets,
> function(x, y) return Intersection(x, y) = []; end);;
gap> Digraph(Petersen);
<digraph with 10 vertices, 30 edges>
gap> b := BinaryRelationOnPoints(
> [[3], [1, 3, 5], [1], [1, 2, 4], [2, 3, 5]]);
Binary Relation on 5 points
gap> gr := Digraph(b);
<digraph with 5 vertices, 11 edges>
gap> gr := Digraph([1 .. 10], ReturnTrue);
<digraph with 10 vertices, 100 edges>
```

The next example illustrates the uses of the fourth and fifth variants of this constructor. The resulting digraph is a strongly regular graph, and it is actually the point graph of the van Lint-Schrijver partial geometry, [vLS81]. The algebraic description is taken from the seminal paper of Calderbank and Kantor [CK86].

Example
```
gap> f := GF(3^4);
GF(3^4)
gap> gamma := First(f, x -> Order(x) = 5);
Z(3^4)^64
gap> L := Union([Zero(f)], List(Group(gamma)));
[ 0*Z(3), Z(3)^0, Z(3^4)^16, Z(3^4)^32, Z(3^4)^48, Z(3^4)^64 ]
gap> omega := Union(List(L, x -> List(Difference(L, [x]), y -> x - y)));
[ Z(3)^0, Z(3), Z(3^4)^5, Z(3^4)^7, Z(3^4)^8, Z(3^4)^13, Z(3^4)^15,
  Z(3^4)^16, Z(3^4)^21, Z(3^4)^23, Z(3^4)^24, Z(3^4)^29, Z(3^4)^31,
  Z(3^4)^32, Z(3^4)^37, Z(3^4)^39, Z(3^4)^45, Z(3^4)^47, Z(3^4)^48,
  Z(3^4)^53, Z(3^4)^55, Z(3^4)^56, Z(3^4)^61, Z(3^4)^63, Z(3^4)^64,
  Z(3^4)^69, Z(3^4)^71, Z(3^4)^72, Z(3^4)^77, Z(3^4)^79 ]
gap> adj := function(x, y)
> return x - y in omega;
> end;
function( x, y ) ... end
gap> digraph := Digraph(AsList(f), adj);
<digraph with 81 vertices, 2430 edges>
gap> group := Group(Z(3));
<group with 1 generators>
```

```
gap> act := \*;
<Operation "*">
gap> digraph := Digraph(group, List(f), act, adj);
<digraph with 81 vertices, 2430 edges>
```

3.1.6 DigraphByAdjacencyMatrix

▷ DigraphByAdjacencyMatrix(*adj*) (operation)

Returns: A digraph.

If *adj* is the adjacency matrix of a digraph in the sense of AdjacencyMatrix (5.2.1), then this operation returns the digraph which is defined by *adj*.

Alternatively, if *adj* is a square boolean matrix, then this operation returns the digraph with Length(*adj*) vertices which has the edge [i,j] if and only if *adj*[i][j] is true.

Example
```
gap> DigraphByAdjacencyMatrix([
> [0, 1, 0, 2, 0],
> [1, 1, 1, 0, 1],
> [0, 3, 2, 1, 1],
> [0, 0, 1, 0, 1],
> [2, 0, 0, 0, 0]]);
<multidigraph with 5 vertices, 18 edges>
gap> gr := DigraphByAdjacencyMatrix([
> [true, false, true],
> [false, false, true],
> [false, true, false]]);
<digraph with 3 vertices, 4 edges>
gap> OutNeighbours(gr);
[ [ 1, 3 ], [ 3 ], [ 2 ] ]
```

3.1.7 DigraphByEdges

▷ DigraphByEdges(*edges*[, *n*]) (operation)

Returns: A digraph.

If *edges* is list of pairs of positive integers, then this function returns the digraph with the minimum number of vertices m such that its edges equal *edges*.

If the optional second argument *n* is a positive integer with *n* >= m (with m defined as above), then this function returns the digraph with *n* vertices and edges *edges*.

See DigraphEdges (5.1.3).

Example
```
gap> DigraphByEdges(
> [[1, 3], [2, 1], [2, 3], [2, 5], [3, 6],
>  [4, 6], [5, 2], [5, 4], [5, 6], [6, 6]]);
<digraph with 6 vertices, 10 edges>
gap> DigraphByEdges(
> [[1, 3], [2, 1], [2, 3], [2, 5], [3, 6],
>  [4, 6], [5, 2], [5, 4], [5, 6], [6, 6]], 10);
<digraph with 10 vertices, 10 edges>
```

3.1.8 EdgeOrbitsDigraph

▷ `EdgeOrbitsDigraph(G, edges[, n])` (operation)

Returns: A new digraph.

If *G* is a permutation group, *edges* is an edge or list of edges, and *n* is a non-negative integer such that *G* fixes `[1 .. n]` setwise, then this operation returns a new digraph with *n* vertices and the union of the orbits of the edges in *edges* under the action of the permutation group *G*. An edge in this context is simply a pair of positive integers.

If the optional third argument *n* is not present, then the largest moved point of the permutation group *G* is used by default.

Example
```
gap> digraph := EdgeOrbitsDigraph(Group((1,3), (1,2)(3,4)),
>                                 [[1, 2], [4, 5]], 5);
<digraph with 5 vertices, 12 edges>
gap> OutNeighbours(digraph);
[ [ 2, 4, 5 ], [ 1, 3, 5 ], [ 2, 4, 5 ], [ 1, 3, 5 ], [  ] ]
gap> RepresentativeOutNeighbours(digraph);
[ [ 2, 4, 5 ], [  ] ]
```

3.1.9 DigraphByInNeighbours

▷ `DigraphByInNeighbours(in)` (operation)

▷ `DigraphByInNeighbors(in)` (operation)

Returns: A digraph.

If *in* is a list of lists of positive integers in the range `[1 .. Length(in)]`, then this function returns the digraph with vertices $E^0 =$ `[1 .. Length(in)]`, and edges corresponding to the entries of *in*. More precisely, there is an edge with source vertex `i` and range vertex `j` if `i` is in *in*`[j]`.

If `i` occurs in *in*`[j]` with multiplicity `k`, then there are `k` multiple edges from `i` to `j`.

See `InNeighbours` (5.2.6).

Example
```
gap> gr := DigraphByInNeighbours([
> [2, 5, 8, 10], [2, 3, 4, 5, 6, 8, 9, 10],
> [1], [3, 5, 7, 8, 10], [2, 5, 7], [3, 6, 7, 9, 10], [1, 4],
> [1, 5, 9], [1, 2, 7, 8], [3, 5]]);
<digraph with 10 vertices, 37 edges>
gap> gr := DigraphByInNeighbours([[2, 3, 2], [1], [1, 2, 3]]);
<multidigraph with 3 vertices, 7 edges>
```

3.2 Changing representations

3.2.1 AsBinaryRelation

▷ `AsBinaryRelation(digraph)` (operation)

Returns: A binary relation.

If *digraph* is a digraph with a positive number of vertices n, and no multiple edges, then this operation returns a binary relation on the points `[1..n]`. The pair `[i,j]` is in the binary relation if and only if `[i,j]` is an edge in *digraph*.

Example

```
gap> gr := Digraph( [ [ 3, 2 ], [ 1, 2 ], [ 2 ], [ 3, 4 ] ] );
<digraph with 4 vertices, 7 edges>
gap> AsBinaryRelation(gr);
Binary Relation on 4 points
```

3.2.2 AsDigraph

▷ `AsDigraph(trans[, n])` (operation)

Returns: A digraph.

If *trans* is a transformation and *n* is a non-negative integer, then this function returns the functional digraph with *n* vertices defined by *trans*. See `IsFunctionalDigraph` (6.1.6). Specifically, the graph has *n* edges: for each vertex x, there is a unique edge with source x; this edge has range `x^trans`.

If the optional second argument *n* is not supplied, then the degree of the transformation *trans* is used by default.

Example

```
gap> f := Transformation([7, 10, 10, 1, 7, 9, 10, 4, 2, 3]);
Transformation( [ 7, 10, 10, 1, 7, 9, 10, 4, 2, 3 ] )
gap> AsDigraph(f);
<digraph with 10 vertices, 10 edges>
gap> AsDigraph(f, 4);
<digraph with 4 vertices, 4 edges>
```

3.2.3 Graph

▷ `Graph(digraph)` (operation)

Returns: A Grape package graph.

If *digraph* is a digraph without multiple edges, then this operation returns a Grape package graph that is isomorphic to *digraph*.

If *digraph* is a multidigraph, then since Grape does not support multiple edges, the multiple edges will be reduced to a single edge in the result. In order words, for a multidigraph this operation will return the same as `Graph(DigraphRemoveAllMultipleEdges(digraph))`.

Example

```
gap> Petersen := Graph(SymmetricGroup(5), [[1, 2]], OnSets,
> function(x, y) return Intersection(x, y) = []; end);
rec( adjacencies := [ [ 3, 5, 8 ] ], group := Group([ (1,2,3,5,7)
  (4,6,8,9,10), (2,4)(6,9)(7,10) ]), isGraph := true,
  names := [ [ 1, 2 ], [ 2, 3 ], [ 3, 4 ], [ 1, 3 ], [ 4, 5 ],
      [ 2, 4 ], [ 1, 5 ], [ 3, 5 ], [ 1, 4 ], [ 2, 5 ] ],
  order := 10, representatives := [ 1 ],
  schreierVector := [ -1, 1, 1, 2, 1, 1, 1, 1, 2, 2 ] )
gap> Digraph(Petersen);
<digraph with 10 vertices, 30 edges>
gap> Graph(last);
rec( adjacencies := [ [ 3, 5, 8 ] ], group := Group([ (1,2,3,5,7)
  (4,6,8,9,10), (2,4)(6,9)(7,10) ]), isGraph := true,
  names := [ [ 1, 2 ], [ 2, 3 ], [ 3, 4 ], [ 1, 3 ], [ 4, 5 ],
      [ 2, 4 ], [ 1, 5 ], [ 3, 5 ], [ 1, 4 ], [ 2, 5 ] ],
```

```
  order := 10, representatives := [ 1 ],
  schreierVector := [ -1, 1, 1, 2, 1, 1, 1, 1, 2, 2 ] )
```

3.2.4 AsGraph

▷ `AsGraph(digraph)` (attribute)

Returns: A Grape package graph.

If *digraph* is a digraph, then this method returns the same as `Graph` (3.2.3), except that the result will be stored as a mutable attribute of *digraph*.

If `AsGraph(digraph)` is called subsequently, then the same GAP object will be returned as before.

Example

```
gap> d := Digraph([[1, 2], [3], []]);
<digraph with 3 vertices, 3 edges>
gap> g := AsGraph(d);
rec( adjacencies := [ [ 1, 2 ], [ 3 ], [  ] ], group := Group(()),
  isGraph := true, names := [ 1 .. 3 ], order := 3,
  representatives := [ 1, 2, 3 ], schreierVector := [ -1, -2, -3 ] )
```

3.2.5 AsTransformation

▷ `AsTransformation(digraph)` (attribute)

Returns: A transformation, or `fail`

If *digraph* is a functional digraph, then `AsTransformation` returns the transformation which is defined by *digraph*. See `IsFunctionalDigraph` (6.1.6). Otherwise, `AsTransformation(digraph)` returns `fail`.

If *digraph* is a functional digraph with n vertices, then `AsTransformation(digraph)` will return the transformation `f` of degree at most n where for each $1 \leq i \leq n$, `i ^ f` is equal to the unique out-neighbour of vertex `i` in *digraph*.

Example

```
gap> gr := Digraph([[1], [3], [2]]);
<digraph with 3 vertices, 3 edges>
gap> gr := CycleDigraph(3);
<digraph with 3 vertices, 3 edges>
gap> AsTransformation(gr);
Transformation( [ 2, 3, 1 ] )
gap> AsPermutation(last);
(1,2,3)
gap> gr := Digraph([[2, 3], [], []]);
<digraph with 3 vertices, 2 edges>
gap> AsTransformation(gr);
fail
```

3.3 New digraphs from old

3.3.1 DigraphCopy

▷ `DigraphCopy(digraph)` (operation)

Returns: A digraph.

This function returns a new copy of *digraph*, retaining none of the attributes or properties of *digraph*.

Example

```
gap> gr := CycleDigraph(10);
<digraph with 10 vertices, 10 edges>
gap> DigraphCopy(gr) = gr;
true
```

3.3.2 InducedSubdigraph

▷ `InducedSubdigraph(`*digraph, verts*`)` (operation)

Returns: A digraph.

If *digraph* is a digraph, and *verts* is a subset of the vertices of *digraph*, then this operation returns a digraph constructed from *digraph* by retaining precisely those vertices in *verts*, and those edges whose source and range vertices are both contained in *verts*.

The vertices of the induced subdigraph are `[1..Length(verts)]` but the original vertex labels can be accessed via `DigraphVertexLabels` (5.1.10).

Example

```
gap> gr := Digraph( [ [ 1, 1, 2, 3, 4, 4 ], [ 1, 3, 4 ], [ 3, 1 ],
> [ 1, 1 ] ] );
<multidigraph with 4 vertices, 13 edges>
gap> InducedSubdigraph( gr, [ 1, 3, 4 ] );
<multidigraph with 3 vertices, 9 edges>
gap> DigraphVertices(last);
[ 1 .. 3 ]
```

3.3.3 ReducedDigraph

▷ `ReducedDigraph(`*digraph*`)` (attribute)

Returns: A digraph.

This function returns a digraph isomorphic to the subdigraph of *digraph* induced by the set of non-isolated vertices, i.e. the set of those vertices of *digraph* which are the source or range of some edge in *digraph*. See `InducedSubdigraph` (3.3.2).

The vertex labels of the graph are preserved, so that a vertex in the new digraph can be matched to the corresponding vertex in *digraph*.

Example

```
gap> d := Digraph([ [ 1, 2 ], [  ], [  ], [ 1, 4 ], [  ] ]);
<digraph with 5 vertices, 4 edges>
gap> r := ReducedDigraph(d);
<digraph with 3 vertices, 4 edges>
gap> OutNeighbours(r);
[ [ 1, 3 ], [ 1, 2 ], [  ] ]
gap> DigraphEdges(r); DigraphEdges(d);
[ [ 1, 1 ], [ 1, 3 ], [ 2, 1 ], [ 2, 2 ] ]
[ [ 1, 1 ], [ 1, 2 ], [ 4, 1 ], [ 4, 4 ] ]
gap> DigraphVertexLabel(r, 3);
2
gap> DigraphVertexLabel(r, 2);
4
```

3.3.4 MaximalSymmetricSubdigraph

▷ `MaximalSymmetricSubdigraph(digraph)` (attribute)
▷ `MaximalSymmetricSubdigraphWithoutLoops(digraph)` (attribute)

Returns: A digraph.

If *digraph* is a digraph, then `MaximalSymmetricSubdigraph` returns a symmetric digraph without multiple edges which has the same vertex set as *digraph*, and whose edge list is formed from *digraph* by ignoring the multiplicity of edges, and by ignoring edges `[u,v]` for which there does not exist an edge `[v,u]`.

The digraph returned by `MaximalSymmetricSubdigraphWithoutLoops` is the same, except that loops are removed.

See `IsSymmetricDigraph` (6.1.9), `IsMultiDigraph` (6.1.7), and `DigraphHasLoops` (6.1.1) for more information.

Example
```
gap> gr := Digraph([[2, 2], [1, 3], [4], [3, 1]]);
<multidigraph with 4 vertices, 7 edges>
gap> not IsSymmetricDigraph(gr) and IsMultiDigraph(gr);
true
gap> OutNeighbours(gr);
[ [ 2, 2 ], [ 1, 3 ], [ 4 ], [ 3, 1 ] ]
gap> sym := MaximalSymmetricSubdigraph(gr);
<digraph with 4 vertices, 4 edges>
gap> IsSymmetricDigraph(sym) and not IsMultiDigraph(sym);
true
gap> OutNeighbours(sym);
[ [ 2 ], [ 1 ], [ 4 ], [ 3 ] ]
```

3.3.5 QuotientDigraph

▷ `QuotientDigraph(digraph, p)` (operation)

Returns: A digraph.

If *digraph* is a digraph, and *p* is a partition of the vertices of *digraph*, then this operation returns a new digraph constructed by amalgamating all vertices of *digraph* which lie in the same part of *p*.

A partition of the vertices of *digraph* is a list of non-empty disjoint lists, such that the union of all the sub-lists is equal to the vertex set of *digraph*. In particular, each vertex must appear in precisely one sub-list.

The vertices of *digraph* in part `i` of *p* will become vertex `i` in the quotient, and every edge of *digraph* with source in part `i` and range in part `j` becomes an edge from `i` to `j` in the quotient. In particular, this means that the quotient of a digraph without multiple edges can have multiple edges.

Example
```
gap> gr := Digraph([ [ 2, 1 ], [ 4 ], [ 1 ], [ 1, 3, 4 ] ]);
<digraph with 4 vertices, 7 edges>
gap> DigraphVertices(gr);
[ 1 .. 4 ]
gap> DigraphEdges(gr);
[ [ 1, 2 ], [ 1, 1 ], [ 2, 4 ], [ 3, 1 ], [ 4, 1 ], [ 4, 3 ],
  [ 4, 4 ] ]
gap> p := [[1], [2, 4], [3]];
[ [ 1 ], [ 2, 4 ], [ 3 ] ]
gap> qr := QuotientDigraph(gr, p);
```

```
<multidigraph with 3 vertices, 7 edges>
gap> DigraphVertices(qr);
[ 1 .. 3 ]
gap> DigraphEdges(qr);
[ [ 1, 2 ], [ 1, 1 ], [ 2, 2 ], [ 2, 1 ], [ 2, 3 ], [ 2, 2 ],
  [ 3, 1 ] ]
gap> QuotientDigraph(EmptyDigraph(0), [ ]);
<digraph with 0 vertices, 0 edges>
```

3.3.6 DigraphReverse

▷ `DigraphReverse(`*`digraph`*`)` (operation)

Returns: A digraph.

If *`digraph`* is a digraph, then this operation returns a digraph constructed from *`digraph`* by reversing the orientation of every edge.

Example
```
gap> gr := Digraph( [ [ 3 ], [ 1, 3, 5 ], [ 1 ], [ 1, 2, 4 ],
> [ 2, 3, 5 ] ] );
<digraph with 5 vertices, 11 edges>
gap> DigraphReverse(gr);
<digraph with 5 vertices, 11 edges>
gap> OutNeighbours(last);
[ [ 2, 3, 4 ], [ 4, 5 ], [ 1, 2, 5 ], [ 4 ], [ 2, 5 ] ]
gap> gr := Digraph( 4,
> [ 1, 1, 1, 2, 3, 3, 4, 4 ],
> [ 1, 2, 4, 1, 2, 4, 3, 4 ] );
<digraph with 4 vertices, 8 edges>
gap> gr := DigraphReverse(gr);
<digraph with 4 vertices, 8 edges>
gap> DigraphRange(gr);
[ 1, 2, 1, 3, 4, 1, 3, 4 ]
gap> DigraphSource(gr);
[ 1, 1, 2, 2, 3, 4, 4, 4 ]
```

3.3.7 DigraphDual

▷ `DigraphDual(`*`digraph`*`)` (attribute)

Returns: A digraph.

If *`digraph`* is a digraph without multiple edges, then this returns the *dual* of *`digraph`*. The *dual* is sometimes called the *complement*.

The *dual* of *`digraph`* has the same vertices as *`digraph`*, and there is an edge in the dual from `i` to `j` whenever there is no edge from `i` to `j` in *`digraph`*.

Example
```
gap> gr := Digraph([[2, 3], [], [4, 6], [5], [],
> [7, 8, 9], [], [], []]);
<digraph with 9 vertices, 8 edges>
gap> DigraphDual(gr);
<digraph with 9 vertices, 73 edges>
```

3.3.8 DigraphSymmetricClosure

▷ `DigraphSymmetricClosure(digraph)` (attribute)

Returns: A digraph.

If *digraph* is a digraph, then this attribute gives the minimal symmetric digraph which has the same vertices and contains all the edges of *digraph*. See `IsSymmetricDigraph` (6.1.9). A digraph is symmetric if, whenever there is an edge from `i` to `j`, there is also an edge from `j` to `i`.

Example

```
gap> gr := Digraph([[1,2,3], [2,4], [1], [3,4]]);
<digraph with 4 vertices, 8 edges>
gap> gr1 := DigraphSymmetricClosure(gr);
<digraph with 4 vertices, 11 edges>
gap> IsSymmetricDigraph(gr1);
true
gap> OutNeighbours(gr1);
[ [ 1, 2, 3 ], [ 2, 4, 1 ], [ 1, 4 ], [ 3, 4, 2 ] ]
gap> gr := Digraph([[2, 2], [1]]);
<multidigraph with 2 vertices, 3 edges>
gap> gr1 := DigraphSymmetricClosure(gr);
<multidigraph with 2 vertices, 4 edges>
gap> OutNeighbours(gr1);
[ [ 2, 2 ], [ 1, 1 ] ]
```

3.3.9 DigraphReflexiveTransitiveClosure

▷ `DigraphReflexiveTransitiveClosure(digraph)` (attribute)

▷ `DigraphTransitiveClosure(digraph)` (attribute)

Returns: A digraph.

If *digraph* is a digraph with no multiple edges, then these attributes return the (reflexive) transitive closure of *digraph*.

A digraph is *reflexive* if it has a loop at every vertex, and it is *transitive* if whenever `[i,j]` and `[j,k]` are edges of *digraph*, `[i,k]` is also an edge. The *(reflexive) transitive closure* of a digraph *digraph* is the least (reflexive and) transitive digraph containing *digraph*.

Let n be the number of vertices of *digraph*, and let m be the number of edges. For an arbitrary digraph, these attributes will use a version of the Floyd-Warshall algorithm, with complexity $O(n^3)$. However, for a topologically sortable digraph [see `DigraphTopologicalSort` (5.1.7)], these attributes will use methods with complexity $O(m+n+m \cdot n)$ when this is faster.

Example

```
gap> gr := Digraph( [ [ 4, 6 ], [ 1, 3 ], [  ], [ 5 ], [  ],
> [ 7, 8, 9 ], [  ], [  ], [  ] ] );;
gap> DigraphTransitiveClosure(gr);
<digraph with 9 vertices, 18 edges>
gap> OutNeighbours(last);
[ [ 4, 5, 6, 7, 8, 9 ], [ 1, 3, 4, 5, 6, 7, 8, 9 ], [  ], [ 5 ],
  [  ], [ 7, 8, 9 ], [  ], [  ], [  ] ]
gap> DigraphReflexiveTransitiveClosure(gr);
<digraph with 9 vertices, 27 edges>
gap> OutNeighbours(last);
[ [ 1, 4, 5, 6, 7, 8, 9 ], [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ], [ 3 ],
  [ 4, 5 ], [ 5 ], [ 6, 7, 8, 9 ], [ 7 ], [ 8 ], [ 9 ] ]
```

3.3.10 DigraphReflexiveTransitiveReduction

▷ `DigraphReflexiveTransitiveReduction(digraph)` (operation)
▷ `DigraphTransitiveReduction(digraph)` (operation)

Returns: A digraph.

If *digraph* is a topologically sortable digraph [see `DigraphTopologicalSort` (5.1.7)] with no multiple edges, then these operations return the (reflexive) transitive reduction of *digraph*.

The (reflexive) transitive reduction of such a digraph is the unique least subgraph such that the (reflexive) transitive closure of the subgraph is equal to the (reflexive) transitive closure of *digraph* [see `DigraphReflexiveTransitiveClosure` (3.3.9)]. In order words, it is the least subgraph of *digraph* which retains the same reachability as *digraph*.

Let n be the number of vertices of an arbitrary digraph, and let m be the number of edges. Then these operations use methods with complexity $O(m+n+m \cdot n)$.

Example
```
gap> gr := Digraph( [ [ 1, 2, 3 ], [ 3 ], [ 3 ] ] );;
gap> DigraphHasLoops(gr);
true
gap> gr1 := DigraphReflexiveTransitiveReduction(gr);
<digraph with 3 vertices, 2 edges>
gap> DigraphHasLoops(gr1);
false
gap> OutNeighbours(gr1);
[ [ 2 ], [ 3 ], [  ] ]
gap> gr2 := DigraphTransitiveReduction(gr);
<digraph with 3 vertices, 4 edges>
gap> DigraphHasLoops(gr2);
true
gap> OutNeighbours(gr2);
[ [ 2, 1 ], [ 3 ], [ 3 ] ]
gap> DigraphReflexiveTransitiveClosure(gr)
>  = DigraphReflexiveTransitiveClosure(gr1);
true
gap> DigraphTransitiveClosure(gr)
>  = DigraphTransitiveClosure(gr2);
true
```

3.3.11 DigraphAddVertex

▷ `DigraphAddVertex(digraph[, label])` (operation)

Returns: A digraph.

The operation returns a new digraph constructed from *digraph* by adding a single new vertex.

If the optional second argument *label* is a GAP object, then the new vertex will be labelled *label*.

Example
```
gap> gr := CompleteDigraph(3);
<digraph with 3 vertices, 6 edges>
gap> new := DigraphAddVertex(gr);
<digraph with 4 vertices, 6 edges>
gap> DigraphVertices(new);
[ 1 .. 4 ]
```

```
gap> new := DigraphAddVertex(gr, Group((1,2)));
<digraph with 4 vertices, 6 edges>
gap> DigraphVertexLabels(new);
[ 1, 2, 3, Group([ (1,2) ]) ]
```

3.3.12 DigraphAddVertices

▷ DigraphAddVertices(*digraph*, *m*[, *labels*]) (operation)

Returns: A digraph.

For a non-negative integer *m*, this operation returns a new digraph constructed from *digraph* by adding *m* new vertices.

If the optional third argument *labels* is a list of length *m* consisting of GAP objects, then the new vertices will be labelled according to this list.

Example
```
gap> gr := CompleteDigraph(3);
<digraph with 3 vertices, 6 edges>
gap> new := DigraphAddVertices(gr, 3);
<digraph with 6 vertices, 6 edges>
gap> DigraphVertices(new);
[ 1 .. 6 ]
gap> new := DigraphAddVertices(gr, 2, [ Group( [(1,2)] ), "d" ] );
<digraph with 5 vertices, 6 edges>
gap> DigraphVertexLabels(new);
[ 1, 2, 3, Group([ (1,2) ]), "d" ]
gap> DigraphAddVertices(gr, 0) = gr;
true
```

3.3.13 DigraphAddEdge

▷ DigraphAddEdge(*digraph*, *edge*) (operation)

Returns: A digraph.

If *edge* is a pairs of vertices of *digraph*, then this operation returns a new digraph constructed from *digraph* by adding a new edge with source *edge*[1] and range *edge*[2].

Example
```
gap> gr1 := Digraph( [ [ 2 ], [ 3 ], [  ] ] );
<digraph with 3 vertices, 2 edges>
gap> DigraphEdges(gr1);
[ [ 1, 2 ], [ 2, 3 ] ]
gap> gr2 := DigraphAddEdge(gr1, [ 3, 1 ]);
<digraph with 3 vertices, 3 edges>
gap> DigraphEdges(gr2);
[ [ 1, 2 ], [ 2, 3 ], [ 3, 1 ] ]
gap> gr3 := DigraphAddEdge(gr2, [ 2, 3 ] );
<multidigraph with 3 vertices, 4 edges>
gap> DigraphEdges(gr3);
[ [ 1, 2 ], [ 2, 3 ], [ 2, 3 ], [ 3, 1 ] ]
```

3.3.14 DigraphAddEdgeOrbit

▷ DigraphAddEdgeOrbit(*digraph*, *edge*) (operation)

Returns: A new digraph.

This operation returns a new digraph with the same vertices and edges as *digraph* and with additional edges consisting of the orbit of the edge *edge* under the action of the DigraphGroup (7.2.3) of *digraph*. If *edge* is already an edge in *digraph*, then *digraph* is returns unchanged.

An edge is simply a pair of vertices of *digraph*.

Example
```
gap> gr1 := CayleyDigraph(DihedralGroup(8));
<digraph with 8 vertices, 24 edges>
gap> gr2 := DigraphAddEdgeOrbit(gr1, [1, 8]);
<digraph with 8 vertices, 32 edges>
gap> DigraphEdges(gr1);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 1 ], [ 2, 8 ], [ 2, 6 ],
  [ 3, 5 ], [ 3, 4 ], [ 3, 7 ], [ 4, 6 ], [ 4, 7 ], [ 4, 1 ],
  [ 5, 3 ], [ 5, 2 ], [ 5, 8 ], [ 6, 4 ], [ 6, 5 ], [ 6, 2 ],
  [ 7, 8 ], [ 7, 1 ], [ 7, 3 ], [ 8, 7 ], [ 8, 6 ], [ 8, 5 ] ]
gap> DigraphEdges(gr2);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 8 ], [ 2, 1 ], [ 2, 8 ],
  [ 2, 6 ], [ 2, 3 ], [ 3, 5 ], [ 3, 4 ], [ 3, 7 ], [ 3, 2 ],
  [ 4, 6 ], [ 4, 7 ], [ 4, 1 ], [ 4, 5 ], [ 5, 3 ], [ 5, 2 ],
  [ 5, 8 ], [ 5, 4 ], [ 6, 4 ], [ 6, 5 ], [ 6, 2 ], [ 6, 7 ],
  [ 7, 8 ], [ 7, 1 ], [ 7, 3 ], [ 7, 6 ], [ 8, 7 ], [ 8, 6 ],
  [ 8, 5 ], [ 8, 1 ] ]
gap> gr3 := DigraphRemoveEdgeOrbit(gr2, [1, 8]);
<digraph with 8 vertices, 24 edges>
gap> gr3 = gr1;
true
```

3.3.15 DigraphAddEdges

▷ DigraphAddEdges(*digraph*, *edges*) (operation)

Returns: A digraph.

If *edges* is a (possibly empty) list of pairs of vertices of *digraph*, then this operation returns a new digraph constructed from *digraph* by adding the edges specified by *edges*. More precisely, for every edge in *edges*, a new edge will be added with source edge[1] and range edges[2].

If an edge is included in *edges* with multiplicity k, then it will be added k times.

Example
```
gap> func := function(n)
>  local source, range, i;
>  source := [  ];
>  range  := [  ];
>  for i in [ 1 .. n - 2 ] do
>    Add(source, i);
>    Add(range, i + 1);
>  od;
>  return Digraph( n, source, range );
> end;;
gap> gr := func(1024);
<digraph with 1024 vertices, 1022 edges>
```

```
gap> gr := DigraphAddEdges(gr,
> [ [ 1023, 1024 ], [ 1, 1024 ], [ 1023, 1024 ], [ 1024, 1 ] ] );
<multidigraph with 1024 vertices, 1026 edges>
```

3.3.16 DigraphRemoveVertex

▷ `DigraphRemoveVertex(digraph, v)` (operation)

Returns: A digraph.

If *v* is a vertex of *digraph*, then this operation returns a new digraph constructed from *digraph* by removing vertex *v*, along with any edge whose source or range vertex is *v*.

If *digraph* has `n` vertices, then the vertices of the new digraph are `[1..n-1]`, but the original labels can be accessed via `DigraphVertexLabels` (5.1.10).

Example
```
gap> gr := Digraph( [ "a", "b", "c" ],
> [ "a", "a", "b", "c", "c" ],
> [ "b", "c", "a", "a", "c" ] );
<digraph with 3 vertices, 5 edges>
gap> DigraphVertexLabels(gr);
[ "a", "b", "c" ]
gap> DigraphEdges(gr);
[ [ 1, 2 ], [ 1, 3 ], [ 2, 1 ], [ 3, 1 ], [ 3, 3 ] ]
gap> new := DigraphRemoveVertex(gr, 2);
<digraph with 2 vertices, 3 edges>
gap> DigraphVertexLabels(new);
[ "a", "c" ]
```

3.3.17 DigraphRemoveVertices

▷ `DigraphRemoveVertices (digraph, verts)` (operation)

Returns: A digraph.

If *verts* is a (possibly empty) duplicate-free list of vertices of *digraph*, then this operation returns a new digraph constructed from *digraph* by removing every vertex in *verts*, along with any edge whose source or range vertex is in *verts*.

If *digraph* has `n` vertices, then the vertices of the new digraph are `[1..n-Length(verts)]`, but the original labels can be accessed via `DigraphVertexLabels` (5.1.10).

Example
```
gap> gr := Digraph( [ [ 3 ], [ 1, 3, 5 ], [ 1 ], [ 1, 2, 4 ],
> [ 2, 3, 5 ] ] );
<digraph with 5 vertices, 11 edges>
gap> SetDigraphVertexLabels( gr, [ "a", "b", "c", "d", "e" ] );
gap> new := DigraphRemoveVertices(gr, [ 2, 4 ] );
<digraph with 3 vertices, 4 edges>
gap> DigraphVertexLabels(new);
[ "a", "c", "e" ]
```

3.3.18 DigraphRemoveEdge

▷ `DigraphRemoveEdge(digraph, edge)` (operation)

Returns: A digraph.

If one of the following holds:

- *digraph* is a digraph with no multiple edges, and *edge* is a pair of vertices of *digraph*, or
- *digraph* is any digraph and *edge* is the index of an edge of *digraph*,

then this operation returns a new digraph constructed from *digraph* by removing the edges specified by *edges*. If, in the first case, the pair of vertices *edge* does not specify an edge of *digraph*, then a new copy of *digraph* will be returned.

Example

```
gap> gr := CycleDigraph(250000);
<digraph with 250000 vertices, 250000 edges>
gap> gr := DigraphRemoveEdge(gr, [ 250000, 1 ]);
<digraph with 250000 vertices, 249999 edges>
gap> gr := DigraphRemoveEdge(gr, 10);
<digraph with 250000 vertices, 249998 edges>
```

3.3.19 DigraphRemoveEdgeOrbit

▷ `DigraphRemoveEdgeOrbit(digraph, edge)` (operation)

Returns: A new digraph.

This operation returns a new digraph with the same vertices as *digraph* and with the orbit of the edge *edge* (under the action of the `DigraphGroup` (7.2.3) of *digraph*) removed. If *edge* is not an edge in *digraph*, then *digraph* is returned unchanged.

An edge is simply a pair of vertices of *digraph*.

Example

```
gap> gr1 := CayleyDigraph(DihedralGroup(8));
<digraph with 8 vertices, 24 edges>
gap> gr2 := DigraphAddEdgeOrbit(gr1, [1, 8]);
<digraph with 8 vertices, 32 edges>
gap> DigraphEdges(gr1);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 1 ], [ 2, 8 ], [ 2, 6 ],
  [ 3, 5 ], [ 3, 4 ], [ 3, 7 ], [ 4, 6 ], [ 4, 7 ], [ 4, 1 ],
  [ 5, 3 ], [ 5, 2 ], [ 5, 8 ], [ 6, 4 ], [ 6, 5 ], [ 6, 2 ],
  [ 7, 8 ], [ 7, 1 ], [ 7, 3 ], [ 8, 7 ], [ 8, 6 ], [ 8, 5 ] ]
gap> DigraphEdges(gr2);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 8 ], [ 2, 1 ], [ 2, 8 ],
  [ 2, 6 ], [ 2, 3 ], [ 3, 5 ], [ 3, 4 ], [ 3, 7 ], [ 3, 2 ],
  [ 4, 6 ], [ 4, 7 ], [ 4, 1 ], [ 4, 5 ], [ 5, 3 ], [ 5, 2 ],
  [ 5, 8 ], [ 5, 4 ], [ 6, 4 ], [ 6, 5 ], [ 6, 2 ], [ 6, 7 ],
  [ 7, 8 ], [ 7, 1 ], [ 7, 3 ], [ 7, 6 ], [ 8, 7 ], [ 8, 6 ],
  [ 8, 5 ], [ 8, 1 ] ]
gap> gr3 := DigraphRemoveEdgeOrbit(gr2, [1, 8]);
<digraph with 8 vertices, 24 edges>
gap> gr3 = gr1;
true
```

3.3.20 DigraphRemoveEdges

▷ `DigraphRemoveEdges(digraph, edges)` (operation)

Returns: A digraph.

If one of the following holds:

- *digraph* is a digraph with no multiple edges, and *edges* is a list of pairs of vertices of *digraph*, or
- *digraph* is any digraph and *edges* is a list of indices of edges of *digraph*,

then this operation returns a new digraph constructed from *digraph* by removing all of the edges specified by *edges* [see DigraphRemoveEdge (3.3.18)].

Example
```
gap> gr := CycleDigraph(250000);
<digraph with 250000 vertices, 250000 edges>
gap> gr := DigraphRemoveEdges(gr, [ [ 250000, 1 ] ]);
<digraph with 250000 vertices, 249999 edges>
gap> gr := DigraphRemoveEdges(gr, [ 10 ]);
<digraph with 250000 vertices, 249998 edges>
```

3.3.21 DigraphRemoveLoops

▷ DigraphRemoveLoops(*digraph*) (operation)

Returns: A digraph.

If *digraph* is a digraph, then this operation returns a new digraph constructed from *digraph* by removing every loop. A loop is an edge with equal source and range.

Example
```
gap> gr := Digraph( [ [ 1, 2, 4 ], [ 1, 4 ], [ 3, 4 ], [ 1, 4, 5 ],
> [ 1, 5 ] ] );
<digraph with 5 vertices, 12 edges>
gap> DigraphRemoveLoops(gr);
<digraph with 5 vertices, 8 edges>
```

3.3.22 DigraphRemoveAllMultipleEdges

▷ DigraphRemoveAllMultipleEdges(*digraph*) (operation)

Returns: A digraph.

If *digraph* is a digraph, then this operation returns a new digraph constructed from *digraph* by removing all multiple edges. The result is the largest subdigraph of *digraph* which does not contain multiple edges.

Example
```
gap> gr1 := Digraph( [ [ 1, 2, 3, 2 ], [ 1, 1, 3 ], [ 2, 2, 2 ] ] );
<multidigraph with 3 vertices, 10 edges>
gap> gr2 := DigraphRemoveAllMultipleEdges(gr1);
<digraph with 3 vertices, 6 edges>
gap> OutNeighbours(gr2);
[ [ 1, 2, 3 ], [ 1, 3 ], [ 2 ] ]
```

3.3.23 DigraphReverseEdges

▷ DigraphReverseEdges(*digraph*, *edges*) (operation)

▷ DigraphReverseEdge(*digraph*, *edge*) (operation)

Returns: A digraph.

If *digraph* is a digraph without multiple edges, and *edges* is either:

- a list of pairs of vertices of *digraph* (the entries of each pair corresponding to the source and the range of an edge, respectively),

- a list of positions of elements in the list `DigraphEdges` (5.1.3),

then `DigraphReverseEdges` returns a new digraph constructed from *digraph* by reversing the orientation of every edge specified by *edges*. If only one edge is to be reversed, then `DigraphReverseEdge` can be used instead. In this case, the second argument should just be a single vertex-pair or a single position.

Note that even though *digraph* cannot have multiple edges, the output may have multiple edges.

Example

```
gap> gr := Digraph( 21,
> [ 1, 1, 1, 5, 7, 9, 11, 21 ],
> [ 7, 2, 8, 21, 19, 1, 2, 1 ]);
<digraph with 21 vertices, 8 edges>
gap> DigraphEdges(gr);
[ [ 1, 7 ], [ 1, 2 ], [ 1, 8 ], [ 5, 21 ], [ 7, 19 ], [ 9, 1 ],
  [ 11, 2 ], [ 21, 1 ] ]
gap> gr2 := DigraphReverseEdges(gr, [ 1, 2, 4 ] );
<digraph with 21 vertices, 8 edges>
gap> gr = DigraphReverseEdges( gr2,
> [ [ 7, 1 ], [ 2, 1 ], [ 21, 5 ] ] );
true
gap> gr2 := DigraphReverseEdge(gr, 5);
<digraph with 21 vertices, 8 edges>
gap> gr2 = DigraphReverseEdge(gr, [ 7, 19 ]);
true
```

3.3.24 DigraphDisjointUnion (for an arbitrary number of digraphs)

▷ `DigraphDisjointUnion(`*gr1, gr2, ...*`)` (function)

▷ `DigraphDisjointUnion(`*list*`)` (function)

Returns: A digraph.

In the first form, if *gr1*, *gr2*, etc. are digraphs, then `DigraphDisjointUnion` returns their disjoint union. In the second form, if *list* is a non-empty list of digraphs, then `DigraphDisjointUnion` returns the disjoint union of the digraphs contained in the list.

For a disjoint union of digraphs, the vertex set is the disjoint union of the vertex sets, and the edge list is the disjoint union of the edge lists.

More specifically, for a collection of digraphs *gr1*, *gr2*, `...`, the disjoint union with have `DigraphNrVertices(`*gr1*`) + DigraphNrVertices(`*gr2*`) + ...` vertices. The edges of *gr1* will remain unchanged, whilst the edges of the ith digraph, *gr*`[i]`, will be changed so that they belong to the vertices of the disjoint union corresponding to *gr*`[i]`. In particular, the edges of *gr*`[i]` will have their source and range increased by `DigraphNrVertices(`*gr1*`) + ... + DigraphNrVertices(`*gr*`[i-1])`.

Note that previously set `DigraphVertexLabels` (5.1.10) will be lost.

Example

```
gap> gr1 := CycleDigraph(3);
<digraph with 3 vertices, 3 edges>
gap> OutNeighbours(gr1);
[ [ 2 ], [ 3 ], [ 1 ] ]
```

```
gap> gr2 := CompleteDigraph(3);
<digraph with 3 vertices, 6 edges>
gap> OutNeighbours(gr2);
[ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ]
gap> union := DigraphDisjointUnion(gr1, gr2);
<digraph with 6 vertices, 9 edges>
gap> OutNeighbours(union);
[ [ 2 ], [ 3 ], [ 1 ], [ 5, 6 ], [ 4, 6 ], [ 4, 5 ] ]
```

3.3.25 DigraphEdgeUnion (for an arbitrary number of digraphs)

▷ `DigraphEdgeUnion(gr1, gr2, ...)` (function)
▷ `DigraphEdgeUnion(list)` (function)

Returns: A digraph.

In the first form, if *gr1*, *gr2*, etc. are digraphs, then `DigraphEdgeUnion` returns their edge union. In the second form, if *list* is a non-empty list of digraphs, then `DigraphEdgeUnion` returns the edge union of the digraphs contained in the list.

The vertex set of the edge union of a collection of digraphs is the *union* of the vertex sets, whilst the edge list of the edge union is the *concatenation* of the edge lists. The number of vertices of the edge union is equal to the *maximum* number of vertices of one of the digraphs, whilst the number of edges of the edge union will equal the *sum* of the number of edges of each digraph.

Note that previously set `DigraphVertexLabels` (5.1.10) will be lost.

Example
```
gap> gr := CycleDigraph(10);
<digraph with 10 vertices, 10 edges>
gap> DigraphEdgeUnion(gr, gr);
<multidigraph with 10 vertices, 20 edges>
gap> gr1 := Digraph( [ [ 2 ], [ 1 ] ] );
<digraph with 2 vertices, 2 edges>
gap> gr2 := Digraph( [ [ 2, 3 ], [ 2 ], [ 1 ] ] );
<digraph with 3 vertices, 4 edges>
gap> union := DigraphEdgeUnion(gr1, gr2);
<multidigraph with 3 vertices, 6 edges>
gap> OutNeighbours(union);
[ [ 2, 2, 3 ], [ 1, 2 ], [ 1 ] ]
gap> union = DigraphByEdges(
> Concatenation(DigraphEdges(gr1), DigraphEdges(gr2)));
true
```

3.3.26 DigraphJoin (for an arbitrary number of digraphs)

▷ `DigraphJoin(gr1, gr2, ...)` (function)
▷ `DigraphJoin(list)` (function)

Returns: A digraph.

In the first form, if *gr1*, *gr2*, etc. are digraphs, then `DigraphJoin` returns their join. In the second form, if *list* is a non-empty list of digraphs, then `DigraphJoin` returns the join of the digraphs contained in the list.

The join of a collection of digraphs *gr1*, *gr2*, ... is formed by first taking the `DigraphDisjointUnion` (3.3.24) of the collection. In the disjoint union, if $i \neq j$ then there are no edges between vertices corresponding to digraphs *gr*[i] and *gr*[j] in the collection; the join is created by including all such edges.

For example, the join of two empty digraphs is a complete bipartite digraph.

Note that previously set DigraphVertexLabels (5.1.10) will be lost.

Example

```
gap> gr := CompleteDigraph(3);
<digraph with 3 vertices, 6 edges>
gap> IsCompleteDigraph(DigraphJoin(gr, gr));
true
gap> gr2 := CycleDigraph(3);
<digraph with 3 vertices, 3 edges>
gap> DigraphJoin(gr, gr2);
<digraph with 6 vertices, 27 edges>
```

3.3.27 LineDigraph

▷ LineDigraph(*digraph*) (operation)

▷ EdgeDigraph(*digraph*) (operation)

Returns: A digraph.

Given a digraph *digraph*, the operation returns the digraph obtained by associating a vertex with each edge of *digraph*, and creating an edge from a vertex v to a vertex u if and only if the terminal vertex of the edge associated with v is the start vertex of the edge associated with u.

Example

```
gap> LineDigraph(CompleteDigraph(3));
<digraph with 6 vertices, 12 edges>
gap> LineDigraph(ChainDigraph(3));
<digraph with 2 vertices, 1 edge>
```

3.3.28 LineUndirectedDigraph

▷ LineUndirectedDigraph(*digraph*) (operation)

▷ EdgeUndirectedDigraph(*digraph*) (operation)

Returns: A digraph.

Given a symmetric digraph *digraph*, the operation returns the symmetric digraph obtained by associating a vertex with each edge of *digraph*, ignoring directions and multiplicites, and adding an edge between two vertices if and only if the corresponding edges have a vertex in common.

Example

```
gap> LineUndirectedDigraph(CompleteDigraph(3));
<digraph with 3 vertices, 6 edges>
gap> LineUndirectedDigraph(DigraphSymmetricClosure(ChainDigraph(3)));
<digraph with 2 vertices, 2 edges>
```

3.3.29 DoubleDigraph

▷ DoubleDigraph(*digraph*) (operation)

Returns: A digraph.

Let *digraph* be a digraph with vertex set V. This function returns the double digraph of *digraph*. The vertex set of the double digraph is the orginal vertex set together with a duplicate. The edges are [u_1,v_2] and [u_2,v_1] if and only if [u,v] is an edge in *digraph*, together with the original edges and their duplicates.

Example

```
gap> out := [[2],[3],[1]];
[ [ 2 ], [ 3 ], [ 1 ] ]
gap> gamma := Digraph(out);
<digraph with 3 vertices, 3 edges>
gap> DoubleDigraph(gamma);
<digraph with 6 vertices, 12 edges>
```

3.3.30 BipartiteDoubleDigraph

▷ BipartiteDoubleDigraph(*digraph*) (operation)

Returns: A digraph.

Let *digraph* be a digraph with vertex set V. This function returns the bipartite double digraph of *digraph*. The vertex set of the double digraph is the orginal vertex set together with a duplicate. The edges are [u_1,v_2] and [u_2,v_1] if and only if [u,v] is an edge in *digraph*. The resulting graph is bipartite, since the orignal edges are not included in the resulting digraph.

Example

```
gap> out := [[2],[3],[1]];
[ [ 2 ], [ 3 ], [ 1 ] ]
gap> gamma := Digraph(out);
<digraph with 3 vertices, 3 edges>
gap> BipartiteDoubleDigraph(gamma);
<digraph with 6 vertices, 6 edges>
```

3.3.31 DigraphAddAllLoops

▷ DigraphAddAllLoops(*digraph*) (operation)

Returns: A digraph.

For a digraph *digraph* this operation return a copy of *digraph* such that a loop is added for every vertex which did not have a loop in *digraph*.

Example

```
gap> gr := EmptyDigraph(13);
<digraph with 13 vertices, 0 edges>
gap> gr := DigraphAddAllLoops(gr);
<digraph with 13 vertices, 13 edges>
gap> OutNeighbours(gr);
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ], [ 6 ], [ 7 ], [ 8 ], [ 9 ],
  [ 10 ], [ 11 ], [ 12 ], [ 13 ] ]
gap> gr := Digraph([ [1, 2, 3], [1, 3], [1] ]);
<digraph with 3 vertices, 6 edges>
gap> gr := DigraphAddAllLoops(gr);
<digraph with 3 vertices, 8 edges>
gap> OutNeighbours(gr);
[ [ 1, 2, 3 ], [ 1, 3, 2 ], [ 1, 3 ] ]
```

3.3.32 DistanceDigraph (for digraph and int)

▷ DistanceDigraph(*digraph*, *i*) (operation)
▷ DistanceDigraph(*digraph*, *list*) (operation)

Returns: A digraph.

The first argument is a digraph, the second argument is a non-negative integer or a list of positive integers. This operation returns a digraph on the same set of vertices as *digraph*, with two vertices being adjacent if and only if the distance between them in *digraph* equals *i* or is a number in *list*. See DigraphShortestDistance (5.3.2).

Example
```
gap> out := [ [ 16, 18, 25 ], [ 17, 20, 25 ], [ 16, 21, 28 ],
>   [ 17, 19, 28 ], [ 17, 24, 26 ], [ 18, 22, 26 ],
>   [ 18, 19, 23 ], [ 19, 27, 29 ], [ 20, 21, 23 ],
>   [ 21, 26, 29 ], [ 20, 22, 27 ], [ 22, 28, 30 ],
>   [ 23, 24, 30 ], [ 16, 24, 27 ], [ 25, 29, 30 ],
>   [ 1, 3, 14 ], [ 2, 4, 5 ], [ 1, 6, 7 ], [ 4, 7, 8 ],
>   [ 2, 9, 11 ], [ 3, 9, 10 ], [ 6, 11, 12 ], [ 7, 9, 13 ],
>   [ 5, 13, 14 ], [ 1, 2, 15 ], [ 5, 6, 10 ], [ 8, 11, 14 ],
>   [ 3, 4, 12 ], [ 8, 10, 15 ], [ 12, 13, 15 ] ];;
gap> digraph := Digraph(out);
<digraph with 30 vertices, 90 edges>
gap> DistanceDigraph(digraph,1);
<digraph with 30 vertices, 90 edges>
gap> DistanceDigraph(digraph,[1,2]);
<digraph with 30 vertices, 270 edges>
```

3.4 Random digraphs

3.4.1 RandomDigraph

▷ RandomDigraph(*n*[, *p*]) (operation)

Returns: A digraph.

If *n* is a positive integer, then this function returns a random digraph with *n* vertices and without multiple edges. The result may or may not have loops.

If the optional second argument *p* is a float with value $0 \leq p \leq 1$, then an edge will exist between each pair of vertices with probability approximately *p*. If *p* is not specified, then a random probability will be assumed (chosen with uniform probability).

Example
```
gap> RandomDigraph(1000);
<digraph with 1000 vertices, 364444 edges>
gap> RandomDigraph(10000, 0.023);
<digraph with 10000 vertices, 2300438 edges>
```

3.4.2 RandomMultiDigraph

▷ RandomMultiDigraph(*n*[, *m*]) (operation)

Returns: A digraph.

If *n* is a positive integer, then this function returns a random digraph with *n* vertices. If the optional second argument *m* is a positive integer, then the digraph will have *m* edges. If *m* is not specified, then the number of edges will be chosen randomly (with uniform probability) from the range `[1 .. ` $\binom{n}{2}$`]`.

The method used by this function chooses each edge from the set of all possible edges with uniform probability. No effort is made to avoid creating multiple edges, so it is possible (but not guaranteed) that the result will have multiple edges. The result may or may not have loops.

Example
```
gap> RandomMultiDigraph(1000);
<multidigraph with 1000 vertices, 216659 edges>
gap> RandomMultiDigraph(1000, 950);
<multidigraph with 1000 vertices, 950 edges>
```

3.4.3 RandomTournament

▷ `RandomTournament(`*n*`)` (operation)

Returns: A digraph.

If *n* is a non-negative integer, this function returns a random tournament with *n* vertices. See `IsTournament` (6.1.10).

Example
```
gap> RandomTournament(10);
<digraph with 10 vertices, 45 edges>
```

3.5 Standard examples

3.5.1 ChainDigraph

▷ `ChainDigraph(`*n*`)` (operation)

Returns: A digraph.

If *n* is a positive integer, this function returns a chain with *n* vertices and *n* `- 1` edges. Specifically, for each vertex `i` (with `i` < `n`), there is a directed edge with source `i` and range `i + 1`.

The `DigraphReflexiveTransitiveClosure` (3.3.9) of a chain represents a total order.

Example
```
gap> ChainDigraph(42);
<digraph with 42 vertices, 41 edges>
```

3.5.2 CompleteDigraph

▷ `CompleteDigraph(`*n*`)` (operation)

Returns: A digraph.

If *n* is a non-negative integer, this function returns the complete digraph with *n* vertices. See `IsCompleteDigraph` (6.1.4).

Example
```
gap> CompleteDigraph(20);
<digraph with 20 vertices, 380 edges>
```

3.5.3 CompleteBipartiteDigraph

▷ `CompleteBipartiteDigraph(m, n)` (operation)

Returns: A digraph.

A complete bipartite digraph is a digraph whose vertices can be partitioned into two non-empty vertex sets, such there exists a unique edge with source `i` and range `j` if and only if `i` and `j` lie in different vertex sets.

If *m* and *n* are positive integers, this function returns the complete bipartite digraph with vertex sets of sizes *m* (containing the vertices `[1 .. m]`) and *n* (containing the vertices `[m + 1 .. m + n]`).

```
gap> CompleteBipartiteDigraph(2, 3);
<digraph with 5 vertices, 12 edges>
```

3.5.4 CycleDigraph

▷ `CycleDigraph(n)` (operation)

Returns: A digraph.

If *n* is a positive integer, this function returns a *cycle* digraph with *n* vertices and *n* edges. Specifically, for each vertex `i` (with `i` < `n`), there is a directed edge with source `i` and range `i + 1`. In addition, there is an edge with source `n` and range `1`.

```
gap> CycleDigraph(1);
<digraph with 1 vertex, 1 edge>
gap> CycleDigraph(123);
<digraph with 123 vertices, 123 edges>
```

3.5.5 EmptyDigraph

▷ `EmptyDigraph(n)` (operation)
▷ `NullDigraph(n)` (operation)

Returns: A digraph.

If *n* is a non-negative integer, this function returns the *empty* or *null* digraph with *n* vertices. An empty digraph is one with no edges.

`NullDigraph` is a synonym for `EmptyDigraph`.

```
gap> EmptyDigraph(20);
<digraph with 20 vertices, 0 edges>
gap> NullDigraph(10);
<digraph with 10 vertices, 0 edges>
```

3.5.6 CayleyDigraph

▷ `CayleyDigraph(G[, gens])` (operation)

Returns: A digraph.

Let *G* be any group and let *gens* be a list of elements of *G*. This function returns the Cayley graph of the group with respect *gens*. The vertices are the elements of *G*. There exists an edge from the vertex `u` to the vertex `v` if and only if there exists a generator `g` in *gens* such that `x * g = y`.

If the optional second argument *gens* is not present, then the generators of *G* are used by default.

Example

```
gap> G := DihedralGroup(8);
<pc group of size 8 with 3 generators>
gap> CayleyDigraph(G);
<digraph with 8 vertices, 24 edges>
gap> G := DihedralGroup(IsPermGroup, 8);
Group([ (1,2,3,4), (2,4) ])
gap> CayleyDigraph(G);
<digraph with 8 vertices, 16 edges>
gap> CayleyDigraph(G, [()]);
<digraph with 8 vertices, 8 edges>
```

3.5.7 JohnsonDigraph

▷ `JohnsonDigraph(n, k)` (operation)

Returns: A digraph.

If `n` and `k` are non-negative integers, then this operation returns a symmetric digraph which corresponds to the undirected *Johnson graph* $J(n,k)$.

The *Johnson graph* $J(n,k)$ has vertices given by all the `k`-subsets of the range `[1 .. k]`, and two vertices are connected by an edge iff their intersection has size $k-1$.

Example

```
gap> gr := JohnsonDigraph(3, 1);
<digraph with 3 vertices, 6 edges>
gap> OutNeighbours(gr);
[ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ]
gap> gr := JohnsonDigraph(4, 2);
<digraph with 6 vertices, 24 edges>
gap> OutNeighbours(gr);
[ [ 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> JohnsonDigraph(1,0);
<digraph with 1 vertex, 0 edges>
```

Chapter 4

Operators

4.1 Operators for digraphs

digraph1 = *digraph2*

returns `true` if *digraph1* and *digraph2* have the same vertices, and `DigraphEdges(`*digraph1*`)` = `DigraphEdges(`*digraph2*`)`, up to some re-ordering of the edge lists.

Note that this operator does not compare the vertex labels of *digraph1* and *digraph2*.

digraph1 < *digraph2*

This operator returns `true` if one of the following holds:

- The number n_1 of vertices in *digraph1* is less than the number n_2 of vertices in *digraph2*;
- $n_1 = n_2$, and the number m_1 of edges in *digraph1* is less than the number m_2 of edges in *digraph2*;
- $n_1 = n_2$, $m_1 = m_2$, and `DigraphEdges(`*digraph1*`)` is less than `DigraphEdges(`*digraph2*`)` after having both of these sets have been sorted with respect to the lexicographical order.

Chapter 5

Attributes and operations

5.1 Vertices and edges

5.1.1 DigraphVertices

▷ DigraphVertices(*digraph*) (attribute)

Returns: A list of integers.

Returns the vertices of the digraph *digraph*.

Note that the vertices of a digraph are always a range of positive integers from 1 to the number of vertices of the graph.

Example
```
gap> gr := Digraph( [ "a", "b", "c" ],
> [ "a", "b", "b" ],
> [ "b", "c", "a" ] );
<digraph with 3 vertices, 3 edges>
gap> DigraphVertices(gr);
[ 1 .. 3 ]
gap> gr := Digraph( [ 1, 2, 3, 4, 5, 7 ],
> [ 1, 2, 2, 4, 4 ],
> [ 2, 3, 5, 3, 5 ] );
<digraph with 6 vertices, 5 edges>
gap> DigraphVertices(gr);
[ 1 .. 6 ]
gap> DigraphVertices(RandomDigraph(100));
[ 1 .. 100 ]
```

5.1.2 DigraphNrVertices

▷ DigraphNrVertices(*digraph*) (attribute)

Returns: An integer.

Returns the number of vertices of the digraph *digraph*.

Example
```
gap> gr := Digraph( [ "a", "b", "c" ],
> [ "a", "b", "b" ],
> [ "b", "c", "a" ] );
<digraph with 3 vertices, 3 edges>
gap> DigraphNrVertices(gr);
```

```
3
gap> gr := Digraph( [ 1, 2, 3, 4, 5, 7 ],
> [ 1, 2, 2, 4, 4 ],
> [ 2, 3, 5, 3, 5 ] );
<digraph with 6 vertices, 5 edges>
gap> DigraphNrVertices(gr);
6
gap> DigraphNrVertices(RandomDigraph(100));
100
```

5.1.3 DigraphEdges

▷ `DigraphEdges(`*`digraph`*`)` (attribute)

Returns: A list of lists.

`DigraphEdges` returns a list of edges of the digraph *digraph*, where each edge is a pair of elements of `DigraphVertices` (5.1.1) of the form `[source,range]`.

The entries of `DigraphEdges(`*`digraph`*`)` are in one-to-one corresponence with the edges of *digraph*. Hence `DigraphEdges(`*`digraph`*`)` is duplicate-free if and only if *digraph* contains no multiple edges.

The entries of `DigraphEdges` are guaranteed to be sorted by their first component (i.e. by the source of each edge), but they are not necessarily then sorted by the second component.

Example
```
gap> gr := Digraph([[1, 3, 4, 3, 5], [1, 2, 3, 5], [2, 4, 5],
> [2, 4, 5], [1]]);
<multidigraph with 5 vertices, 16 edges>
gap> DigraphEdges(gr);
[ [ 1, 1 ], [ 1, 3 ], [ 1, 4 ], [ 1, 3 ], [ 1, 5 ], [ 2, 1 ],
  [ 2, 2 ], [ 2, 3 ], [ 2, 5 ], [ 3, 2 ], [ 3, 4 ], [ 3, 5 ],
  [ 4, 2 ], [ 4, 4 ], [ 4, 5 ], [ 5, 1 ] ]
```

5.1.4 DigraphNrEdges

▷ `DigraphNrEdges(`*`digraph`*`)` (attribute)

Returns: An integer.

This function returns the number of edges of the digraph *digraph*.

Example
```
gap> gr := Digraph([[1, 3, 4, 5], [1, 2, 3, 5], [2, 4, 5],
> [2, 4, 5], [1]]);;
gap> DigraphNrEdges(gr);
15
gap> gr := Digraph(["a", "b", "c"],
> ["a", "b", "b"],
> ["b", "a", "a"]);
<multidigraph with 3 vertices, 3 edges>
gap> DigraphNrEdges(gr);
3
```

5.1.5 DigraphSinks

▷ `DigraphSinks(digraph)` (attribute)

Returns: A list of vertices.

This function returns a list of the sinks of the digraph *digraph*. A sink of a digraph is a vertex with out-degree zero. See `OutDegreeOfVertex` (5.2.9).

Example
```
gap> gr := Digraph( [ [ 3, 5, 2, 2 ], [ 3 ], [  ], [ 5, 2, 5, 3 ], [  ] ] );
<multidigraph with 5 vertices, 9 edges>
gap> DigraphSinks(gr);
[ 3, 5 ]
```

5.1.6 DigraphSources

▷ `DigraphSources(digraph)` (attribute)

Returns: A list of vertices.

This function returns a list of the sources of the digraph *digraph*. A source of a digraph is a vertex with in-degree zero. See `InDegreeOfVertex` (5.2.11).

Example
```
gap> gr := Digraph( [ [ 3, 5, 2, 2 ], [ 3 ], [  ], [ 5, 2, 5, 3 ], [  ] ] );
<multidigraph with 5 vertices, 9 edges>
gap> DigraphSources(gr);
[ 1, 4 ]
```

5.1.7 DigraphTopologicalSort

▷ `DigraphTopologicalSort(digraph)` (attribute)

Returns: A list of positive integers, or `fail`.

If *digraph* is a digraph with no cycles of length greater than 1, then `DigraphTopologicalSort` returns the vertices of *digraph* ordered so that every edge's source appears no earlier in the list than its range. If the digraph *digraph* contains cycles of length greater than 1, then this operation returns `fail`.

The method used for this attribute has complexity $O(m+n)$ where m is the number of edges (counting multiple edges as one) and n is the number of vertices in the digraph.

Example
```
gap> gr := Digraph([[2, 3], [], [4, 6], [5], [], [7, 8, 9], [],
> [], []]);
<digraph with 9 vertices, 8 edges>
gap> DigraphTopologicalSort(gr);
[ 2, 5, 4, 7, 8, 9, 6, 3, 1 ]
```

5.1.8 DigraphBicomponents

▷ `DigraphBicomponents(digraph)` (attribute)

Returns: A pair of lists of vertices, or `fail`.

If *digraph* is a bipartite digraph, i.e. if it satisfies `IsBipartiteDigraph` (6.1.3), then `DigraphBicomponents` returns a pair of bicomponents of *digraph*. Otherwise, `DigraphBicomponents` returns `fail`.

For a bipartite digraph, the vertices can be partitioned into two non-empty sets such that the source and range of any edge are in distinct sets. The parts of this partition are called *bicomponents* of *digraph*. Equivalently, a pair of bicomponents of *digraph* consists of the color-classes of a 2-coloring of *digraph*.

For a bipartite digraph with at least 3 vertices, there is a unique pair of bicomponents of bipartite if and only if the digraph is connected. See `IsConnectedDigraph` (6.3.2) for more information.

Example
```
gap> gr := CycleDigraph(3);
<digraph with 3 vertices, 3 edges>
gap> DigraphBicomponents(gr);
fail
gap> gr := ChainDigraph(5);
<digraph with 5 vertices, 4 edges>
gap> DigraphBicomponents(gr);
[ [ 1, 3, 5 ], [ 2, 4 ] ]
gap> gr := Digraph([[5], [1, 4], [5], [5], []]);
<digraph with 5 vertices, 5 edges>
gap> DigraphBicomponents(gr);
[ [ 1, 3, 4 ], [ 2, 5 ] ]
```

5.1.9 DigraphVertexLabel

▷ `DigraphVertexLabel(`*digraph, i*`)` (operation)
▷ `SetDigraphVertexLabel(`*digraph, i, obj*`)` (operation)

If *digraph* is a digraph, then the first operation returns the label of the vertex *i*. The second operation can be used to set the label of the vertex *i* in *digraph* to the arbitrary GAP object *obj*.

The label of a vertex can be changed an arbitrary number of times. If no label has been set for the vertex *i*, then the default value is *i*.

If *digraph* is a digraph created from a record with a component `vertices`, then the labels of the vertices are set to the value of this component.

Induced subdigraphs, and other operations which create new digraphs from old ones, inherit their labels from their parents.

Example
```
gap> gr := Digraph([[3], [1, 3, 5], [1], [1, 2, 4], [2, 3, 5]]);
<digraph with 5 vertices, 11 edges>
gap> DigraphVertexLabel(gr, 3);
3
gap> gr := Digraph(["a", "b", "c"], [], []);
<digraph with 3 vertices, 0 edges>
gap> DigraphVertexLabel(gr, 2);
"b"
gap> SetDigraphVertexLabel(gr, 2, "d");
gap> DigraphVertexLabel(gr, 2);
"d"
gap> gr := InducedSubdigraph(gr, [1, 2]);
<digraph with 2 vertices, 0 edges>
gap> DigraphVertexLabel(gr, 2);
"d"
```

5.1.10 DigraphVertexLabels

▷ DigraphVertexLabels(*digraph*) (operation)
▷ SetDigraphVertexLabels(*digraph*, *list*) (operation)

If *digraph* is a digraph, then DigraphVertexLabels returns a copy of the labels of the vertices in *digraph*. SetDigraphVertexLabels can be used to set the labels of the vertices in *digraph* to the list of arbitrary GAP objects *list*.

The label of a vertex can be changed an arbitrary number of times. If no label has been set for the vertex *i*, then the default value is *i*.

If *digraph* is a digraph created from a record with a component vertices, then the labels of the vertices are set to the value of this component.

Induced subdigraphs, and other operations which create new digraphs from old ones, inherit their labels from their parents.

Example
```
gap> gr := Digraph([[3], [1, 3, 5], [1], [1, 2, 4], [2, 3, 5]]);
<digraph with 5 vertices, 11 edges>
gap> DigraphVertexLabels(gr);
[ 1 .. 5 ]
gap> gr := Digraph(["a", "b", "c"], [], []);
<digraph with 3 vertices, 0 edges>
gap> DigraphVertexLabels(gr);
[ "a", "b", "c" ]
gap> SetDigraphVertexLabel(gr, 2, "d");
gap> DigraphVertexLabels(gr);
[ "a", "d", "c" ]
gap> gr := InducedSubdigraph(gr, [1, 3]);
<digraph with 2 vertices, 0 edges>
gap> DigraphVertexLabels(gr);
[ "a", "c" ]
```

5.1.11 DigraphInEdges

▷ DigraphInEdges(*digraph*, *vertex*) (operation)

Returns: A list of edges.

DigraphInEdges returns the list of all edges of *digraph* which have *vertex* as their range.

Example
```
gap> gr := Digraph( [ [ 2, 2 ], [ 3, 3 ], [ 4, 4 ], [ 1, 1 ] ] );
<multidigraph with 4 vertices, 8 edges>
gap> DigraphInEdges(gr, 2);
[ [ 1, 2 ], [ 1, 2 ] ]
```

5.1.12 DigraphOutEdges

▷ DigraphOutEdges(*digraph*, *vertex*) (operation)

Returns: A list of edges.

DigraphOutEdges returns the list of all edges of *digraph* which have *vertex* as their source.

Example
```
gap> gr := Digraph( [ [ 2, 2 ], [ 3, 3 ], [ 4, 4 ], [ 1, 1 ] ] );
<multidigraph with 4 vertices, 8 edges>
```

```
gap> DigraphOutEdges(gr, 2);
[ [ 2, 3 ], [ 2, 3 ] ]
```

5.1.13 IsDigraphEdge (for digraph and list)

▷ IsDigraphEdge(*digraph*, *list*) (operation)
▷ IsDigraphEdge(*digraph*, *u*, *v*) (operation)

Returns: `true` or `false`.

In the first form, this function returns `true` if and only if the list *list* specifies an edge in the digraph *digraph*. Specifically, this operation returns `true` if *list* is a pair of positive integers where *list*[1] is the source and *list*[2] is the range of an edge in *digraph*, and `false` otherwise.

The second form simply returns `true` if [*u*, *v*] is an edge in *digraph*, and `false` otherwise.

Example

```
gap> gr := Digraph(6, [1, 1, 2, 4, 6], [2, 2, 6, 3, 1]);
<multidigraph with 6 vertices, 5 edges>
gap> IsDigraphEdge(gr, [1, 1]);
false
gap> IsDigraphEdge(gr, [1, 2]);
true
gap> IsDigraphEdge(gr, [1, 8]);
false
```

5.2 Neighbours and degree

5.2.1 AdjacencyMatrix

▷ AdjacencyMatrix(*digraph*) (attribute)

Returns: A square matrix of non-negative integers.

This function returns the adjacency matrix `mat` of the digraph *digraph*. The value of the matrix entry `mat[i][j]` is the number of edges in *digraph* with source `i` and range `j`.

Example

```
gap> gr := Digraph([[2, 2, 2], [1, 3, 6, 8, 9, 10], [4, 6, 8],
> [1, 2, 3, 9], [3, 3], [3, 5, 6, 10], [1, 2, 7],
> [1, 2, 3, 10, 5, 6, 10], [1, 3, 4, 5, 8, 10],
> [2, 3, 4, 6, 7, 10]]);
<multidigraph with 10 vertices, 44 edges>
gap> mat := AdjacencyMatrix(gr);;
gap> Display(mat);
[ [  0,  3,  0,  0,  0,  0,  0,  0,  0,  0 ],
  [  1,  0,  1,  0,  0,  1,  0,  1,  1,  1 ],
  [  0,  0,  0,  1,  0,  1,  0,  1,  0,  0 ],
  [  1,  1,  1,  0,  0,  0,  0,  0,  1,  0 ],
  [  0,  0,  2,  0,  0,  0,  0,  0,  0,  0 ],
  [  0,  0,  1,  0,  1,  1,  0,  0,  0,  1 ],
  [  1,  1,  0,  0,  0,  0,  1,  0,  0,  0 ],
  [  1,  1,  1,  0,  1,  1,  0,  0,  0,  2 ],
  [  1,  0,  1,  1,  1,  0,  0,  1,  0,  1 ],
  [  0,  1,  1,  1,  0,  1,  1,  0,  0,  1 ] ]
```

5.2.2 BooleanAdjacencyMatrix

▷ `BooleanAdjacencyMatrix(digraph)` (attribute)

Returns: A square matrix of booleans.

If *digraph* is a digraph with a positive number of vertices `n`, then `BooleanAdjacencyMatrix(digraph)` returns the boolean adjacency matrix `mat` of *digraph*. The value of the matrix entry `mat[j][i]` is `true` if and only if there exists an edge in *digraph* with source `j` and range `i`.

Note this the boolean adjacency loses information about multiple edges.

If *digraph* has no vertices, then this attribute returns the empty list.

Example

```
gap> gr := Digraph([[3, 4], [2, 3], [1, 2, 4], [4]]);
<digraph with 4 vertices, 8 edges>
gap> BooleanAdjacencyMatrix(gr);
[ [ false, false, true, true ], [ false, true, true, false ],
  [ true, true, false, true ], [ false, false, false, true ] ]
gap> gr := CycleDigraph(4);;
gap> BooleanAdjacencyMatrix(gr);
[ [ false, true, false, false ], [ false, false, true, false ],
  [ false, false, false, true ], [ true, false, false, false ] ]
gap> BooleanAdjacencyMatrix(EmptyDigraph(0));
[  ]
```

5.2.3 DigraphAdjacencyFunction

▷ `DigraphAdjacencyFunction(digraph)` (attribute)

Returns: A function.

If *digraph* is a digraph, then `DigraphAdjacencyFunction` returns a function which takes two integer parameters `x, y` and returns `true` if there exists an edge from vertex `x` to vertex `y` in *digraph* and `false` if not.

Example

```
gap> digraph := Digraph([[1, 2], [3], []]);
<digraph with 3 vertices, 3 edges>
gap> foo := DigraphAdjacencyFunction(digraph);
function( u, v ) ... end
gap> foo(1, 1);
true
gap> foo(1, 2);
true
gap> foo(1, 3);
false
gap> foo(3, 1);
false
gap> gr := Digraph(["a", "b", "c"],
>                  ["a", "b", "b"],
>                  ["b", "a", "a"]);
<multidigraph with 3 vertices, 3 edges>
gap> foo := DigraphAdjacencyFunction(gr);
function( u, v ) ... end
gap> foo(1, 2);
true
```

```
gap> foo(3, 2);
false
gap> foo(3, 1);
false
```

5.2.4 DigraphRange

▷ `DigraphRange(digraph)` (attribute)
▷ `DigraphSource(digraph)` (attribute)

Returns: A list of positive integers.

`DigraphRange` and `DigraphSource` return the range and source of the digraph *digraph*. More precisely, position `i` in `DigraphRange(digraph)` is the range of the `i`th edge of *digraph*.

Example
```
gap> gr := Digraph([[1, 2, 3, 5], [1, 3, 4], [2, 3],
> [2], [1, 2, 3, 4]]);
<digraph with 5 vertices, 14 edges>
gap> DigraphRange(gr);
[ 1, 2, 3, 5, 1, 3, 4, 2, 3, 2, 1, 2, 3, 4 ]
gap> DigraphSource(gr);
[ 1, 1, 1, 1, 2, 2, 2, 3, 3, 4, 5, 5, 5, 5 ]
gap> DigraphEdges(gr);
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 5 ], [ 2, 1 ], [ 2, 3 ],
  [ 2, 4 ], [ 3, 2 ], [ 3, 3 ], [ 4, 2 ], [ 5, 1 ], [ 5, 2 ],
  [ 5, 3 ], [ 5, 4 ] ]
```

5.2.5 OutNeighbours

▷ `OutNeighbours(digraph)` (attribute)
▷ `OutNeighbors(digraph)` (attribute)
▷ `OutNeighboursCopy(digraph)` (operation)
▷ `OutNeighborsCopy(digraph)` (operation)

Returns: The adjacencies of a digraph.

This function returns the list `out` of out-neighbours of each vertex of the digraph *digraph*. More specifically, a vertex `j` appears in `out[i]` each time there exists an edge with source `i` and range `j` in *digraph*.

The function `OutNeighbours` returns an immutable list of immutable lists, whereas the function `OutNeighboursCopy` returns a copy of `OutNeighbours` which is a mutable list of mutable lists.

Example
```
gap> gr := Digraph(["a", "b", "c"],
> ["a", "b", "b"],
> ["b", "a", "c"]);
<digraph with 3 vertices, 3 edges>
gap> OutNeighbours(gr);
[ [ 2 ], [ 1, 3 ], [  ] ]
gap> gr := Digraph(3,
> [1, 2, 3, 1, 1, 2],
> [1, 2, 3, 2, 3, 1]);
<digraph with 3 vertices, 6 edges>
gap> OutNeighbours(gr);
```

```
[ [ 1, 2, 3 ], [ 2, 1 ], [ 3 ] ]
gap> gr := Digraph(3,
> [1, 2, 3, 1, 1, 2, 1],
> [1, 2, 3, 2, 3, 1, 2]);
<multidigraph with 3 vertices, 7 edges>
gap> OutNeighbours(gr);
[ [ 1, 2, 3, 2 ], [ 2, 1 ], [ 3 ] ]
gap> OutNeighboursCopy(gr);
[ [ 1, 2, 3, 2 ], [ 2, 1 ], [ 3 ] ]
```

5.2.6 InNeighbours

▷ `InNeighbours(`*`digraph`*`)` (attribute)

▷ `InNeighbors(`*`digraph`*`)` (attribute)

Returns: A list of lists of vertices.

This function returns the list `inn` of in-neighbours of each vertex of the digraph *`digraph`*. More specifically, a vertex `j` appears in `inn[i]` each time there exists an edge with source `j` and range `i` in *`digraph`*.

Note that each entry of `inn` is sorted into ascending order.

Example

```
gap> gr := Digraph(["a", "b", "c"],
> ["a", "b", "b"],
> ["b", "a", "c"]);
<digraph with 3 vertices, 3 edges>
gap> InNeighbours(gr);
[ [ 2 ], [ 1 ], [ 2 ] ]
gap> gr := Digraph(3,
> [1, 2, 3, 1, 1, 2],
> [1, 2, 3, 2, 3, 1]);
<digraph with 3 vertices, 6 edges>
gap> InNeighbours(gr);
[ [ 1, 2 ], [ 1, 2 ], [ 1, 3 ] ]
gap> gr := Digraph(3,
> [1, 2, 3, 1, 1, 2, 1],
> [1, 2, 3, 2, 3, 1, 2]);
<multidigraph with 3 vertices, 7 edges>
gap> InNeighbours(gr);
[ [ 1, 2 ], [ 1, 1, 2 ], [ 1, 3 ] ]
```

5.2.7 OutDegrees

▷ `OutDegrees(`*`digraph`*`)` (attribute)

▷ `OutDegreeSequence(`*`digraph`*`)` (attribute)

▷ `OutDegreeSet(`*`digraph`*`)` (attribute)

Returns: A list of non-negative integers.

Given a digraph *`digraph`* with n vertices, the function `OutDegrees` returns a list `out` of length n, such that for a vertex `i` in *`digraph`*, the value of `out[i]` is the out-degree of vertex `i`. See `OutDegreeOfVertex` (5.2.9).

The function `OutDegreeSequence` returns the same list, after it has been sorted into non-increasing order.

The function `OutDegreeSet` returns the same list, sorted into increasing order with duplicate entries removed.

Example
```
gap> gr := Digraph([[1, 3, 2, 2], [], [2, 1], []]);
<multidigraph with 4 vertices, 6 edges>
gap> OutDegrees(gr);
[ 4, 0, 2, 0 ]
gap> OutDegreeSequence(gr);
[ 4, 2, 0, 0 ]
gap> OutDegreeSet(gr);
[ 0, 2, 4 ]
```

5.2.8 InDegrees

▷ `InDegrees(digraph)` (attribute)
▷ `InDegreeSequence(digraph)` (attribute)
▷ `InDegreeSet(digraph)` (attribute)

Returns: A list of non-negative integers.

Given a digraph *digraph* with n vertices, the function `InDegrees` returns a list `inn` of length n, such that for a vertex `i` in *digraph*, the value of `inn[i]` is the in-degree of vertex `i`. See `InDegreeOfVertex` (5.2.11).

The function `InDegreeSequence` returns the same list, after it has been sorted into non-increasing order.

The function `InDegreeSet` returns the same list, sorted into increasing order with duplicate entries removed.

Example
```
gap> gr := Digraph( [ [ 1, 3, 2, 2, 4 ], [ ], [ 2, 1, 4 ], [ ] ] );
<multidigraph with 4 vertices, 8 edges>
gap> InDegrees(gr);
[ 2, 3, 1, 2 ]
gap> InDegreeSequence(gr);
[ 3, 2, 2, 1 ]
gap> InDegreeSet(gr);
[ 1, 2, 3 ]
```

5.2.9 OutDegreeOfVertex

▷ `OutDegreeOfVertex(digraph, vertex)` (operation)

Returns: The non-negative integer.

This operation returns the out-degree of the vertex *vertex* in the digraph *digraph*. The out-degree of *vertex* is the number of edges in *digraph* whose source is *vertex*.

Example
```
gap> gr := Digraph( [ [ 2, 2, 1 ], [ 1, 4 ], [ 2, 2, 4, 2 ],
> [ 1, 1, 2, 2, 1, 2, 2 ] ] );
<multidigraph with 4 vertices, 16 edges>
gap> OutDegreeOfVertex(gr, 1);
3
```

```
gap> OutDegreeOfVertex(gr, 2);
2
gap> OutDegreeOfVertex(gr, 3);
4
gap> OutDegreeOfVertex(gr, 4);
7
```

5.2.10 OutNeighboursOfVertex

▷ OutNeighboursOfVertex(*digraph, vertex*) (operation)
▷ OutNeighborsOfVertex(*digraph, vertex*) (operation)

Returns: A list of vertices.

This operation returns the list out of vertices of the digraph *digraph*. A vertex i appears in the list out each time there exists an edge with source *vertex* and range i in *digraph*; in particular, this means that out may contain duplicates.

Example
```
gap> gr := Digraph( [ [ 2, 2, 3 ], [ 1, 3, 4 ], [ 2, 2, 3 ],
> [ 1, 1, 2, 2, 1, 2, 2 ] ] );
<multidigraph with 4 vertices, 16 edges>
gap> OutNeighboursOfVertex(gr, 1);
[ 2, 2, 3 ]
gap> OutNeighboursOfVertex(gr, 3);
[ 2, 2, 3 ]
```

5.2.11 InDegreeOfVertex

▷ InDegreeOfVertex(*digraph, vertex*) (operation)

Returns: A non-negative integer.

This operation returns the in-degree of the vertex *vertex* in the digraph *digraph*. The in-degree of *vertex* is the number of edges in *digraph* whose range is *vertex*.

Example
```
gap> gr := Digraph( [ [ 2, 2, 1 ], [ 1, 4 ], [ 2, 2, 4, 2 ],
> [ 1, 1, 2, 2, 1, 2, 2 ] ] );
<multidigraph with 4 vertices, 16 edges>
gap> InDegreeOfVertex(gr, 1);
5
gap> InDegreeOfVertex(gr, 2);
9
gap> InDegreeOfVertex(gr, 3);
0
gap> InDegreeOfVertex(gr, 4);
2
```

5.2.12 InNeighboursOfVertex

▷ InNeighboursOfVertex(*digraph, vertex*) (operation)
▷ InNeighborsOfVertex(*digraph, vertex*) (operation)

Returns: A list of postitive vertices.

This operation returns the list `inn` of vertices of the digraph *digraph*. A vertex `i` appears in the list `inn` each time there exists an edge with source `i` and range *vertex* in *digraph*; in particular, this means that `inn` may contain duplicates.

Example
```
gap> gr := Digraph( [ [ 2, 2, 3 ], [ 1, 3, 4 ], [ 2, 2, 3 ],
> [ 1, 1, 2, 2, 1, 2, 2 ] ] );
<multidigraph with 4 vertices, 16 edges>
gap> InNeighboursOfVertex(gr, 1);
[ 2, 4, 4, 4 ]
gap> InNeighboursOfVertex(gr, 2);
[ 1, 1, 3, 3, 4, 4, 4, 4 ]
```

5.2.13 DigraphLoops

▷ `DigraphLoops(digraph)` (attribute)

Returns: A list of vertices.

If *digraph* is a digraph, then `DigraphLoops` returns the list consisting of the `DigraphVertices` (5.1.1) of *digraph* at which there is a loop. See `DigraphHasLoops` (6.1.1).

Example
```
gap> gr := Digraph([[2], [3], []]);
<digraph with 3 vertices, 2 edges>
gap> DigraphHasLoops(gr);
false
gap> DigraphLoops(gr);
[  ]
gap> gr := Digraph([[3, 5], [1], [2, 4, 3], [4], [2, 1]]);
<digraph with 5 vertices, 9 edges>
gap> DigraphLoops(gr);
[ 3, 4 ]
```

5.3 Reachability and connectivity

5.3.1 DigraphDiameter

▷ `DigraphDiameter(digraph)` (attribute)

Returns: An integer or `fail`.

This function returns the diameter of the digraph *digraph*.

If a digraph *digraph* is strongly connected and has at least 1 vertex, then the *diameter* is the maximum shortest distance between any pair of distinct vertices. Otherwise then the diameter of *digraph* is undefined, and this function returns the value `fail`.

See `DigraphShortestDistances` (5.3.3).

Example
```
gap> gr := Digraph( [ [ 2 ], [ 3 ], [ 4, 5 ], [ 5 ],
> [ 1, 2, 3, 4, 5 ] ] );
<digraph with 5 vertices, 10 edges>
gap> DigraphDiameter(gr);
3
gap> gr := Digraph( [ [ 2 ], [  ] ] );
<digraph with 2 vertices, 1 edge>
```

```
gap> DigraphDiameter(gr);
fail
gap> IsStronglyConnectedDigraph(gr);
false
```

5.3.2 DigraphShortestDistance (for a digraph and two vertices)

▷ DigraphShortestDistance(*digraph*, *u*, *v*) (operation)
▷ DigraphShortestDistance(*digraph*, *list*) (operation)
▷ DigraphShortestDistance(*digraph*, *list1*, *list2*) (operation)

Returns: An integer or `fail`

If there is a path in the digraph *digraph* between vertex *u* and vertex *v*, then this operation returns the length of the shortest such path. If no such path exists, then this operation returns `fail`.

If the second form is used, then *list* should be a list of length two, containing two positive integers which correspond to the vertices *u* and *v*.

Note that as usual a vertex is considered to be at distance 0 from itself.

If the third form is used, then *list1* and *list2* are both lists of vertices. The shortest path between *list1* and *list2* is then the length of the shortest path which starts with a vertex in *list1* and terminates at a vertex in *list2*, if such path exists. If *list1* and *list2* have non-empty intersection, the operation returns `0`.

Example
```
gap> gr := Digraph([ [2], [3], [1,4], [1,3], [5] ]);
<digraph with 5 vertices, 7 edges>
gap> DigraphShortestDistance(gr, 1, 3);
2
gap> DigraphShortestDistance(gr, [3, 3]);
0
gap> DigraphShortestDistance(gr, 5, 2);
fail
gap> DigraphShortestDistance(gr, [1,2], [4, 5]);
2
gap> DigraphShortestDistance(gr, [1,3], [3, 5]);
0
```

5.3.3 DigraphShortestDistances

▷ DigraphShortestDistances(*digraph*) (attribute)

Returns: A square matrix.

If *digraph* is a digraph with n vertices, then this function returns an $n \times n$ matrix `mat`, where each entry is either a non-negative integer or `fail`. If $n = 0$, then an empty list is returned.

If there is a directed path from vertex `i` to vertex `j`, then the value of `mat[i][j]` is the length of the shortest such path. If no such path exists, then the value of `mat[i][j]` is `fail`. We use the convention that the distance from every vertex to itself is `0`, i.e. `mat[i][i] = 0` for all vertices `i`.

The method used in this function is a version of the Floyd-Warshall algorithm, and has complexity $O(n^3)$.

Example
```
gap> gr := Digraph( [ [ 1, 2 ], [ 3 ], [ 1, 2 ], [ 4 ] ] );
<digraph with 4 vertices, 6 edges>
```

```
gap> mat := DigraphShortestDistances(gr);;
gap> Display(mat);
[ [     0,     1,     2,  fail ],
  [     2,     0,     1,  fail ],
  [     1,     1,     0,  fail ],
  [  fail,  fail,  fail,     0 ] ]
```

5.3.4 DigraphLongestDistanceFromVertex

▷ `DigraphLongestDistanceFromVertex(`*`digraph, v`*`)` (operation)

Returns: An integer.

If *digraph* is a digraph and *v* is a vertex in *digraph*, then this operation returns the length of the longest directed walk in *digraph* which begins at vertex *v*.

- If there exists a directed walk starting at vertex *v* which traverses a cycle or a loop, then we consider there to be a walk of infinite length from *v* (realised by repeatedly traversing the loop/cycle), and so the result is `infinity`. To disallow walks using loops, try using `DigraphRemoveLoops` (3.3.21):

  `DigraphLongestDistanceFromVertex(DigraphRemoveLoops(`*`digraph`*`,`*`v`*`))`.

- If no walk from vertex *v* exists, i.e. if *v* is a sink of the digraph (`DigraphSinks` (5.1.5)), then the result is `0`.

- Otherwise, if all directed walks starting at vertex *v* have finite length, then the length of the longest such walk is returned.

Example
```
gap> gr := Digraph( [ [ 2 ], [ 3, 4 ], [  ], [ 5 ], [  ], [ 6 ] ] );
<digraph with 6 vertices, 5 edges>
gap> DigraphLongestDistanceFromVertex(gr, 1);
3
gap> DigraphLongestDistanceFromVertex(gr, 3);
0
gap> 3 in DigraphSinks(gr);
true
gap> DigraphLongestDistanceFromVertex(gr, 6);
infinity
```

5.3.5 DigraphDistanceSet (for a digraph, a pos int, and an int)

▷ `DigraphDistanceSet(`*`digraph, vertex, distance`*`)` (operation)

▷ `DigraphDistanceSet(`*`digraph, vertex, distances`*`)` (operation)

Returns: A list of vertices

This operation returns the list of all vertices in digraph *digraph* such that the shortest distance to a vertex *vertex* is *distance* or is in the list *distances*.

digraph should be a digraph, *vertex* should be a positive integer, *distance* should be a non-negative integer, and *distances* should be a list of non-negative integers.

Example
```
gap> gr := Digraph([ [2], [3], [1,4], [1,3] ]);
<digraph with 4 vertices, 6 edges>
```

```
gap> DigraphDistanceSet(gr, 2, [1,2]);
[ 3, 1, 4 ]
gap> DigraphDistanceSet(gr, 3, 1);
[ 1, 4 ]
gap> DigraphDistanceSet(gr, 2, 0);
[ 2 ]
```

5.3.6 DigraphGirth

▷ `DigraphGirth(digraph)` (attribute)

Returns: An integer or `infinity`.

The *girth* of a digraph is the length of its shortest simple cycle, i.e. the shortest non-trivial directed path starting and ending at the same vertex and passing through no vertex more than once.

If *digraph* has no cycles, then this function will return `infinity`. If *digraph* contains a loop, then this function will return `1`.

In the worst case, the method used in this function is a version of the Floyd-Warshall algorithm, and has complexity `O(n ^ 3)`, where *n* is the number of vertices in *digraph*. If the digraph has known automorphisms [see `DigraphGroup` (7.2.3)], then the performance is likely to be better.

For symmetric digraphs, see also `DigraphUndirectedGirth` (5.3.7).

Example
```
gap> gr := Digraph([[1], [1]]);
<digraph with 2 vertices, 2 edges>
gap> DigraphGirth(gr);
1
gap> gr := Digraph([[2, 3], [3], [4], []]);
<digraph with 4 vertices, 4 edges>
gap> DigraphGirth(gr);
infinity
gap> gr := Digraph([[2, 3], [3], [4], [1]]);
<digraph with 4 vertices, 5 edges>
gap> DigraphGirth(gr);
3
```

5.3.7 DigraphUndirectedGirth

▷ `DigraphUndirectedGirth(digraph)` (attribute)

Returns: An integer or `infinity`.

If *digraph* is a symmetric digraph, then this function returns the girth of *digraph* when treated as an undirected graph (i.e. each pair of edges $[i, j]$ and $[j, i]$ is treated as a single edge between i and j).

The *girth* of an undirected graph is the length of its shortest simple cycle, i.e. the shortest non-trivial path starting and ending at the same vertex and passing through no vertex more than once.

If *digraph* has no cycles, then this function will return `infinity`.

Example
```
gap> gr := Digraph([[2,4], [1,3], [2,4], [1,3]]);
<digraph with 4 vertices, 8 edges>
gap> DigraphUndirectedGirth(gr);
4
gap> gr := Digraph([[2], [1,3], [2]]);
```

```
<digraph with 3 vertices, 4 edges>
gap> DigraphUndirectedGirth(gr);
infinity
gap> gr := Digraph([[1], [], [4], [3]]);
<digraph with 4 vertices, 3 edges>
gap> DigraphUndirectedGirth(gr);
1
```

5.3.8 DigraphConnectedComponents

▷ `DigraphConnectedComponents(digraph)` (attribute)

Returns: A record.

This function returns the record `wcc` corresponding to the weakly connected components of the digraph *digraph*. Two vertices of *digraph* are in the same weakly connected component whenever they are equal, or there exists a path (ignoring the orientation of edges) between them. In other words, two vertices are in the same weakly connected component of *digraph* if and only if they are in the same strongly connected component (see `DigraphStronglyConnectedComponents` (5.3.10)) of the `DigraphSymmetricClosure` (3.3.8) of *digraph*.

The set of weakly connected components is a partition of the vertex set of *digraph*.

The record `wcc` has 2 components: `comps` and `id`. The component `comps` is a list of the weakly connected components of *digraph* (each of which is a list of vertices). The component `id` is a list such that the vertex `i` is an element of the weakly connected component `comps[id[i]]`.

The method used in this function has complexity $O(m+n)$, where m is the number of edges and n is the number of vertices in the digraph.

Example

```
gap> gr := Digraph( [ "a", "b", "c" ],
> [ "a", "b", "b" ],
> [ "b", "c", "a" ] );
<digraph with 3 vertices, 3 edges>
gap> DigraphConnectedComponents(gr);
rec( comps := [ [ 1, 2, 3 ] ], id := [ 1, 1, 1 ] )
gap> gr := Digraph( [ [ 1 ], [ 1, 2 ], [  ] ] );
<digraph with 3 vertices, 3 edges>
gap> DigraphConnectedComponents(gr);
rec( comps := [ [ 1, 2 ], [ 3 ] ], id := [ 1, 1, 2 ] )
gap> gr := Digraph( [ ] );
<digraph with 0 vertices, 0 edges>
gap> DigraphConnectedComponents(gr);
rec( comps := [  ], id := [  ] )
```

5.3.9 DigraphConnectedComponent

▷ `DigraphConnectedComponent(digraph, vertex)` (operation)

Returns: A list of vertices.

If *vertex* is a vertex in the digraph *digraph*, then this operation returns the connected component of *vertex* in *digraph*. See `DigraphConnectedComponents` (5.3.8) for more information.

Example

```
gap> gr := Digraph( [ [ 3 ], [ 2 ], [ 1, 2 ], [ 4 ] ] );
<digraph with 4 vertices, 5 edges>
```

```
gap> DigraphConnectedComponent(gr, 3);
[ 1, 2, 3 ]
gap> DigraphConnectedComponent(gr, 2);
[ 1, 2, 3 ]
gap> DigraphConnectedComponent(gr, 4);
[ 4 ]
```

5.3.10 DigraphStronglyConnectedComponents

▷ DigraphStronglyConnectedComponents(*digraph*) (attribute)

Returns: A record.

This function returns the record `scc` corresponding to the strongly connected components of the digraph *digraph*. Two vertices of *digraph* are in the same strongly connected component whenever they are equal, or there is a directed path from each vertex to the other. The set of strongly connected components is a partition of the vertex set of *digraph*.

The record `scc` has 2 components: `comps` and `id`. The component `comps` is a list of the strongly connected components of *digraph* (each of which is a list of vertices). The component `id` is a list such that the vertex `i` is an element of the strongly connected component `comps[id[i]]`.

The method used in this function is a non-recursive version of Gabow's Algorithm [Gab00] and has complexity $O(m+n)$ where m is the number of edges (counting multiple edges as one) and n is the number of vertices in the digraph.

Example
```
gap> gr := Digraph( [ "a", "b", "c" ],
> [ "a", "b", "b" ],
> [ "b", "c", "a" ] );
<digraph with 3 vertices, 3 edges>
gap> DigraphStronglyConnectedComponents(gr);
rec( comps := [ [ 3 ], [ 1, 2 ] ], id := [ 2, 2, 1 ] )
```

5.3.11 DigraphStronglyConnectedComponent

▷ DigraphStronglyConnectedComponent(*digraph*, *vertex*) (operation)

Returns: A list of vertices.

If *vertex* is a vertex in the digraph *digraph*, then this operation returns the strongly connected component of *vertex* in *digraph*. See DigraphStronglyConnectedComponents (5.3.10) for more information.

Example
```
gap> gr := Digraph( [ [ 3 ], [ 2 ], [ 1, 2 ], [ 3 ] ] );
<digraph with 4 vertices, 5 edges>
gap> DigraphStronglyConnectedComponent(gr, 3);
[ 1, 3 ]
gap> DigraphStronglyConnectedComponent(gr, 2);
[ 2 ]
gap> DigraphStronglyConnectedComponent(gr, 4);
[ 4 ]
```

5.3.12 DigraphPeriod

▷ `DigraphPeriod(digraph)` (attribute)

Returns: An integer.

This function returns the period of the digraph *digraph*.

If a digraph *digraph* has at least one cycle, then the period is the greatest positive integer which divides the lengths of all cycles of *digraph*. If *digraph* has no cycles, then this function returns 0.

A digraph with a period of 1 is said to be *aperiodic*. See `IsAperiodicDigraph` (6.3.4).

Example
```
gap> gr := Digraph( [ [ 6 ], [ 1 ], [ 2 ], [ 3 ], [ 4, 4 ], [ 5 ] ] );
<multidigraph with 6 vertices, 7 edges>
gap> DigraphPeriod(gr);
6
gap> gr := Digraph( [ [ 2 ], [ 3, 5 ], [ 4 ], [ 5 ], [ 1, 2 ] ] );
<digraph with 5 vertices, 7 edges>
gap> DigraphPeriod(gr);
1
gap> gr := Digraph( [ [ 2 ], [ ] ] );
<digraph with 2 vertices, 1 edge>
gap> DigraphPeriod(gr);
0
gap> IsAcyclicDigraph(gr);
true
```

5.3.13 DigraphFloydWarshall

▷ `DigraphFloydWarshall(digraph, func, nopath, edge)` (operation)

Returns: A matrix.

If *digraph* is a digraph with n vertices, then this operation returns an $n \times n$ matrix `mat` containing the output of a generalised version of the Floyd-Warshall algorithm, applied to *digraph*.

The operation `DigraphFloydWarshall` is customised by the arguments *func*, *nopath*, and *edge*. The arguments *nopath* and *edge* can be arbitrary GAP objects. The argument *func* must be a function which accepts 4 arguments: the matrix `mat`, followed by 3 postive integers. The function *func* is where the work to calculate the desired outcome must be performed.

This method initialises the matrix `mat` by setting entry `mat[i][j]` to equal *edge* if there is an edge with source `i` and range `j`, and by setting entry `mat[i][j]` to equal *nopath* otherwise. The final part of `DigraphFloydWarshall` then calls the function *func* inside three nested for loops, over the vertices of *digraph*:

```
for i in DigraphsVertices(digraph) do
  for j in DigraphsVertices(digraph) do
    for k in DigraphsVertices(digraph) do
      func(mat, i, j, k);
    od;
  od;
od;
```

The matrix `mat` is then returned as the result. An example of using `DigraphFloydWarshall` to calculate the shortest (non-zero) distances between the vertices of a digraph is shown below:

Example

```
gap> gr := Digraph( [ [ 5 ], [ 3, 6 ], [ 2, 5 ], [ 1, 4, 5 ],
> [ 4, 6 ], [ 5, 6 ] ] );
<digraph with 6 vertices, 12 edges>
gap> func := function(mat, i, j, k)
>   if mat[i][k] <> -1 and mat[k][j] <> -1 then
>     if (mat[i][j] = -1) or (mat[i][j] > mat[i][k] + mat[k][j]) then
>       mat[i][j] := mat[i][k] + mat[k][j];
>     fi;
>   fi;
> end;
function( mat, i, j, k ) ... end
gap> shortest_distances := DigraphFloydWarshall( gr, func, -1, 1 );;
gap> Display(shortest_distances);
[ [   3,  -1,  -1,   2,   1,   2 ],
  [   4,   2,   1,   3,   2,   1 ],
  [   3,   1,   2,   2,   1,   2 ],
  [   1,  -1,  -1,   1,   1,   2 ],
  [   2,  -1,  -1,   1,   2,   1 ],
  [   3,  -1,  -1,   2,   1,   1 ] ]
```

5.3.14 IsReachable

▷ `IsReachable(`*`digraph, u, v`*`)` (operation)

Returns: `true` or `false`.

This operation returns `true` if there exists a directed path (of non-zero length) from vertex *u* to vertex *v* in the digraph *digraph*, and `false` if there does not exist such a path.

The method for `IsReachable` has worst case complexity of $O(m+n)$ where m is the number of edges and n the number of vertices in *digraph*.

Example

```
gap> gr := Digraph( [ [ 2 ], [ 3 ], [ 2, 3 ] ] );
<digraph with 3 vertices, 4 edges>
gap> IsReachable(gr, 1, 3);
true
gap> IsReachable(gr, 2, 1);
false
gap> IsReachable(gr, 3, 3);
true
gap> IsReachable(gr, 1, 1);
false
```

5.3.15 DigraphAllSimpleCircuits

▷ `DigraphAllSimpleCircuits(`*`digraph`*`)` (attribute)

Returns: A list of lists of vertices.

If *digraph* is a digraph, then `DigraphAllSimpleCircuits` returns a list of the *simple circuits* in *digraph*.

A *simple circuit* in *digraph* is a non-trivial directed path in *digraph* from a vertex `v` back to `v`, in which no vertex is visited more than once. Since a circuit of length `n` is determined by its first `n` vertices, a circuit $v_1 \to \cdots \to v_n \to v_1$ can be represented as the list $[v_1, \ldots, v_n]$. For each simple circuit of *digraph*, `DigraphAllSimpleCircuits(digraph)` will include precisely one such list to represents the circuit (cyclic permutations of such a list describe the same circuit).

Note that a loop is a simple circuit.

Example
```
gap> gr := Digraph([[], [3], [2, 4], [5, 4], [4]]);
<digraph with 5 vertices, 6 edges>
gap> DigraphAllSimpleCircuits(gr);
[ [ 4 ], [ 4, 5 ], [ 2, 3 ] ]
gap> gr := ChainDigraph(10);;
gap> DigraphAllSimpleCircuits(gr);
[  ]
gap> gr := Digraph([[3], [1], [1]]);
<digraph with 3 vertices, 3 edges>
gap> DigraphAllSimpleCircuits(gr);
[ [ 1, 3 ] ]
```

5.3.16 DigraphLongestSimpleCircuit

▷ `DigraphLongestSimpleCircuit(digraph)` (attribute)

Returns: A list of vertices or `fail`.

If *digraph* is a digraph, then `DigraphLongestSimpleCircuit` returns the longest *simple circuit* in *digraph*.

A *simple circuit* in *digraph* is a non-trivial directed path in *digraph* from a vertex `v` back to `v`, in which no vertex is visited more than once. Since a circuit of length `n` is determined by its first `n` vertices, a circuit $v_1 \to \cdots \to v_n \to v_1$ can be represented as the list $[v_1, \ldots, v_n]$. Note that a loop is a simple circuit.

If *digraph* has no simple circuits, then this attribute returns `fail`. If *digraph* has multiple simple circuits of maximum length, then this attribute returns one of them.

See `DigraphAllSimpleCircuits` (5.3.15).

Example
```
gap> gr := Digraph([[], [3], [2, 4], [5, 4], [4]]);;
gap> DigraphLongestSimpleCircuit(gr);
[ 4, 5 ]
gap> gr := ChainDigraph(10);;
gap> DigraphLongestSimpleCircuit(gr);
fail
gap> gr := Digraph([[3], [1], [1, 4], [1, 1]]);;
gap> DigraphLongestSimpleCircuit(gr);
[ 1, 3, 4 ]
```

5.3.17 DigraphLayers

▷ `DigraphLayers(digraph, vertex)` (operation)

Returns: A list.

This operation returns a list `list` such that `list[i]` is the list of vertices whose minimum distance from the vertex *vertex* in *digraph* is `i - 1`. Vertex *vertex* is assumed to be at distance `0` from itself.

Example

```
gap> gr := CompleteDigraph(4);;
gap> DigraphLayers(gr,1);
[ [ 1 ], [ 2, 3, 4 ] ]
```

5.3.18 DigraphDegeneracy

▷ DigraphDegeneracy(*digraph*) (attribute)

Returns: A non-negative integer, or `fail`.

If *digraph* is a symmetric digraph without multiple edges - see `IsSymmetricDigraph` (6.1.9) and `IsMultiDigraph` (6.1.7) - then this attribute returns the degeneracy of *digraph*.

The degeneracy of a digraph is the least integer `k` such that every induced of *digraph* contains a vertex whose number of neighbours (excluding itself) is at most `k`. Note that this means that loops are ignored.

If *digraph* is not symmetric or has multiple edges then this attribute returns `fail`.

Example

```
gap> gr := DigraphSymmetricClosure(ChainDigraph(5));;
gap> DigraphDegeneracy(gr);
1
gap> gr := CompleteDigraph(5);;
gap> DigraphDegeneracy(gr);
4
gap> gr := Digraph([[1], [2, 4, 5], [3, 4], [2, 3, 4], [2], []]);
<digraph with 6 vertices, 10 edges>
gap> DigraphDegeneracy(gr);
1
```

5.3.19 DigraphDegeneracyOrdering

▷ DigraphDegeneracyOrdering(*digraph*) (attribute)

Returns: A list of integers, or `fail`.

If *digraph* is a digraph for which `DigraphDegeneracy(`*digraph*`)` is a non-negative integer `k` - see `DigraphDegeneracy` (5.3.18) - then this attribute returns a degeneracy ordering of the vertices of the vertices of *digraph*.

A degeneracy ordering of *digraph* is a list `ordering` of the vertices of *digraph* ordered such that for any position `i` of the list, the vertex `ordering[i]` has at most `k` neighbours in later position of the list.

If `DigraphDegeneracy(`*digraph*`)` returns `fail`, then this attribute returns `fail`.

Example

```
gap> gr := DigraphSymmetricClosure(ChainDigraph(5));;
gap> DigraphDegeneracyOrdering(gr);
[ 5, 4, 3, 2, 1 ]
gap> gr := CompleteDigraph(5);;
gap> DigraphDegeneracyOrdering(gr);
[ 5, 4, 3, 2, 1 ]
gap> gr := Digraph([[1], [2, 4, 5], [3, 4], [2, 3, 4], [2], []]);
<digraph with 6 vertices, 10 edges>
gap> DigraphDegeneracyOrdering(gr);
[ 1, 6, 5, 2, 4, 3 ]
```

Chapter 6

Properties of digraphs

6.1 Edge properties

6.1.1 DigraphHasLoops

▷ `DigraphHasLoops(`*digraph*`)` (property)

Returns: `true` or `false`.

Returns `true` if the digraph *digraph* is has loops, and `false` if it does not. A loop is an edge with equal source and range.

Example

```
gap> gr := Digraph( [ [ 1, 2 ], [ 2 ] ] );
<digraph with 2 vertices, 3 edges>
gap> DigraphEdges(gr);
[ [ 1, 1 ], [ 1, 2 ], [ 2, 2 ] ]
gap> DigraphHasLoops(gr);
true
gap> gr := Digraph( [ [ 2, 3 ], [ 1 ], [ 2 ] ] );
<digraph with 3 vertices, 4 edges>
gap> DigraphEdges(gr);
[ [ 1, 2 ], [ 1, 3 ], [ 2, 1 ], [ 3, 2 ] ]
gap> DigraphHasLoops(gr);
false
```

6.1.2 IsAntisymmetricDigraph

▷ `IsAntisymmetricDigraph(`*digraph*`)` (property)

Returns: `true` or `false`.

This property is `true` if the digraph *digraph* is antisymmetric, and `false` if it is not.

A digraph is *antisymmetric* if whenever there is an edge with source `u` and range `v`, and an edge with source `v` and range `u`, then the vertices `u` and `v` are equal.

Example

```
gap> gr1 := Digraph( [ [ 2 ], [ 1, 3 ], [ 2, 3 ] ] );
<digraph with 3 vertices, 5 edges>
gap> IsAntisymmetricDigraph(gr1);
false
gap> DigraphEdges(gr1){[ 1, 2 ]};
[ [ 1, 2 ], [ 2, 1 ] ]
```

```
gap> gr2 := Digraph( [ [ 1, 2 ], [ 3, 3 ], [ 1 ] ] );
<multidigraph with 3 vertices, 5 edges>
gap> IsAntisymmetricDigraph(gr2);
true
gap> DigraphEdges(gr2);
[ [ 1, 1 ], [ 1, 2 ], [ 2, 3 ], [ 2, 3 ], [ 3, 1 ] ]
```

6.1.3 IsBipartiteDigraph

▷ `IsBipartiteDigraph(`*`digraph`*`)` (property)

Returns: `true` or `false`.

This property is `true` if the digraph *`digraph`* is bipartite, and `false` if it is not. A digraph is bipartite if and only if the vertices of *`digraph`* can be partitioned into two non-empty sets such that the source and range of any edge of *`digraph`* lie in distinct sets. Equivalently, a digraph is bipartite if and only if it is 2-colorable; see `DigraphColoring` (7.3.9).

See also `DigraphBicomponents` (5.1.8).

Example
```
gap> gr := ChainDigraph(4);
<digraph with 4 vertices, 3 edges>
gap> IsBipartiteDigraph(gr);
true
gap> gr := CycleDigraph(3);
<digraph with 3 vertices, 3 edges>
gap> IsBipartiteDigraph(gr);
false
```

6.1.4 IsCompleteDigraph

▷ `IsCompleteDigraph(`*`digraph`*`)` (property)

Returns: `true` or `false`.

Returns `true` if the digraph *`digraph`* is complete, and `false` if it is not.

A digraph is *complete* if it has no loops, and for all *distinct* vertices `i` and `j`, there is exactly one edge with source `i` and range `j`. Equivalently, a digraph with n vertices is complete precisely when it has $n(n-1)$ edges, no loops, and no multiple edges.

Example
```
gap> gr := Digraph( [ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ] );
<digraph with 3 vertices, 6 edges>
gap> IsCompleteDigraph(gr);
true
gap> gr := Digraph( [ [ 2, 2 ], [ 1 ] ] );
<multidigraph with 2 vertices, 3 edges>
gap> IsCompleteDigraph(gr);
false
```

6.1.5 IsEmptyDigraph

▷ `IsEmptyDigraph(`*`digraph`*`)` (property)

▷ `IsNullDigraph(`*`digraph`*`)` (property)

Returns: `true` or `false`.

Returns `true` if the digraph *digraph* is empty, and `false` if it is not. A digraph is *empty* if it has no edges.

`IsNullDigraph` is a synonym for `IsEmptyDigraph`.

Example
```
gap> gr := Digraph( [ [  ], [  ] ] );
<digraph with 2 vertices, 0 edges>
gap> IsEmptyDigraph(gr);
true
gap> IsNullDigraph(gr);
true
gap> gr := Digraph( [ [  ], [ 1 ] ] );
<digraph with 2 vertices, 1 edge>
gap> IsEmptyDigraph(gr);
false
gap> IsNullDigraph(gr);
false
```

6.1.6 IsFunctionalDigraph

▷ `IsFunctionalDigraph(digraph)` (property)

Returns: `true` or `false`.

This property is `true` if the digraph *digraph* is functional.

A digraph is *functional* if every vertex is the source of a unique edge.

Example
```
gap> gr1 := Digraph( [ [ 3 ], [ 2 ], [ 2 ], [ 1 ], [ 6 ], [ 5 ] ] );
<digraph with 6 vertices, 6 edges>
gap> IsFunctionalDigraph(gr1);
true
gap> gr2 := Digraph( [ [ 1, 2 ], [ 1 ] ] );
<digraph with 2 vertices, 3 edges>
gap> IsFunctionalDigraph(gr2);
false
gap> gr3 := Digraph( 3, [ 1, 2, 3 ], [ 2, 3, 1 ] );
<digraph with 3 vertices, 3 edges>
gap> IsFunctionalDigraph(gr3);
true
```

6.1.7 IsMultiDigraph

▷ `IsMultiDigraph(digraph)` (property)

Returns: `true` or `false`.

A *multidigraph* is one that has at least two edges with equal source and range.

Example
```
gap> gr := Digraph(["a", "b", "c"], ["a", "b", "b"], ["b", "c", "a"]);
<digraph with 3 vertices, 3 edges>
gap> IsMultiDigraph(gr);
false
gap> gr := Digraph(3, [1, 2, 3, 1, 1, 2], [1, 2, 3, 2, 3, 1]);
<digraph with 3 vertices, 6 edges>
gap> IsMultiDigraph(gr);
false
```

```
gap> gr := Digraph(3, [1, 2, 3, 1, 1, 2, 1], [1, 2, 3, 2, 3, 1, 2]);
<multidigraph with 3 vertices, 7 edges>
gap> IsMultiDigraph(gr);
true
```

6.1.8 IsReflexiveDigraph

▷ IsReflexiveDigraph(*digraph*) (property)

Returns: true or false.

This property is true if the digraph *digraph* is reflexive, and false if it is not. A digraph is *reflexive* if it has a loop at every vertex.

Example
```
gap> gr := Digraph( [ [ 1, 2 ], [ 2 ] ] );
<digraph with 2 vertices, 3 edges>
gap> IsReflexiveDigraph(gr);
true
gap> gr := Digraph( rec ( nrvertices := 4,
> source := [ 1, 1, 2, 2, 3, 4, 4 ],
> range := [ 3, 1, 4, 2, 3, 2, 1 ] )  );
<digraph with 4 vertices, 7 edges>
gap> IsReflexiveDigraph(gr);
false
```

6.1.9 IsSymmetricDigraph

▷ IsSymmetricDigraph(*digraph*) (property)

Returns: true or false.

This property is true if the digraph *digraph* is symmetric, and false if it is not.

A *symmetric digraph* is one where for each non-loop edge, having source u and range v, there is a corresponding edge with source v and range u. If there are n edges with source u and range v, then there must be precisely n edges with source v and range u. In other words, an undirected digraph has a symmetric adjacency matrix AdjacencyMatrix (5.2.1).

Example
```
gap> gr1 := Digraph( [ [ 2 ], [ 1, 3 ], [ 2, 3 ] ] );
<digraph with 3 vertices, 5 edges>
gap> IsSymmetricDigraph(gr1);
true
gap> adj1 := AdjacencyMatrix(gr1);;
gap> Display(adj1);
[ [  0,  1,  0 ],
  [  1,  0,  1 ],
  [  0,  1,  1 ] ]
gap> adj1 = TransposedMat(adj1);
true
gap> gr1 = DigraphReverse(gr1);
true
gap> gr2 := Digraph( [ [ 2, 3 ], [ 1, 3 ], [ 2, 3 ] ] );
<digraph with 3 vertices, 6 edges>
gap> IsSymmetricDigraph(gr2);
false
```

```
gap> adj2 := AdjacencyMatrix(gr2);;
gap> Display(adj2);
[ [  0,  1,  1 ],
  [  1,  0,  1 ],
  [  0,  1,  1 ] ]
gap> adj2 = TransposedMat(adj2);
false
```

6.1.10 IsTournament

▷ `IsTournament(digraph)` (property)

Returns: `true` or `false`.

This property is `true` if the digraph *digraph* is a tournament, and `false` if it is not.

A tournament is an orientation of a complete (undirected) graph. Specifically, a tournament is a digraph which has a unique directed edge (of some orientation) between any pair of distinct vertices, and no loops.

Example
```
gap> gr := Digraph( [ [ 2, 3, 4 ], [ 3, 4 ], [ 4 ], [  ] ] );
<digraph with 4 vertices, 6 edges>
gap> IsTournament(gr);
true
gap> gr := Digraph( [ [ 2 ], [ 1 ], [ 3 ] ] );
<digraph with 3 vertices, 3 edges>
gap> IsTournament(gr);
false
```

6.1.11 IsTransitiveDigraph

▷ `IsTransitiveDigraph(digraph)` (property)

Returns: `true` or `false`.

This property is `true` if the digraph *digraph* is transitive, and `false` if it is not. A digraph is *transitive* if whenever `[ i, j ]` and `[ j, k ]` are edges of the digraph, then `[ i, k ]` is also an edge of the digraph.

Let n be the number of vertices of an arbitrary digraph, and let m be the number of edges. For general digraphs, the methods used for this property use a version of the Floyd-Warshall algorithm, and have complexity $O(n^3)$. However for digraphs which are topologically sortable [`DigraphTopologicalSort` (5.1.7)], then methods with complexity $O(m+n+m\cdot n)$ will be used when appropriate.

Example
```
gap> gr := Digraph( [ [ 1, 2 ], [ 3 ], [ 3 ] ] );
<digraph with 3 vertices, 4 edges>
gap> IsTransitiveDigraph(gr);
false
gap> gr2 := Digraph( [ [ 1, 2, 3 ], [ 3 ], [ 3 ] ] );
<digraph with 3 vertices, 5 edges>
gap> IsTransitiveDigraph(gr2);
true
gap> gr2 = DigraphTransitiveClosure(gr);
true
```

```
gap> gr3 := Digraph( [ [ 1, 2, 2, 3 ], [ 3, 3 ], [ 3 ] ] );
<multidigraph with 3 vertices, 7 edges>
gap> IsTransitiveDigraph(gr3);
true
```

6.2 Regularity

6.2.1 IsInRegularDigraph

▷ `IsInRegularDigraph(`*`digraph`*`)` (property)

Returns: `true` or `false`.

This property is `true` if there is an integer `n` such that for every vertex `v` of digraph *`digraph`* there are exactly `n` edges terminating in `v`. See also `IsOutRegularDigraph` (6.2.2) and `IsRegularDigraph` (6.2.3).

Example
```
gap> IsInRegularDigraph(CompleteDigraph(4));
true
gap> IsInRegularDigraph(ChainDigraph(4));
false
```

6.2.2 IsOutRegularDigraph

▷ `IsOutRegularDigraph(`*`digraph`*`)` (property)

Returns: `true` or `false`.

This property is `true` if there is an integer `n` such that for every vertex `v` of digraph *`digraph`* there are exactly `n` edges starting at `v`. See also `IsInRegularDigraph` (6.2.1) and `IsRegularDigraph` (6.2.3).

Example
```
gap> IsOutRegularDigraph(CompleteDigraph(4));
true
gap> IsOutRegularDigraph(ChainDigraph(4));
false
```

6.2.3 IsRegularDigraph

▷ `IsRegularDigraph(`*`digraph`*`)` (property)

Returns: `true` or `false`.

This property is `true` if there is an integer `n` such that for every vertex `v` of digraph *`digraph`* there are exactly `n` edges starting and terminating at `v`. In other words, the property is `true` if *`digraph`* is both in-regular and and out-regular. See also `IsInRegularDigraph` (6.2.1) and `IsOutRegularDigraph` (6.2.2).

Example
```
gap> IsRegularDigraph(CompleteDigraph(4));
true
gap> IsRegularDigraph(ChainDigraph(4));
false
```

6.2.4 IsDistanceRegularDigraph

▷ `IsDistanceRegularDigraph(`*digraph*`)` (property)

Returns: `true` or `false`.

If *digraph* is a connected symmetric graph, this property returns `true` if for any two vertices `u` and `v` of *digraph* and any two integers `i` and `j` between `0` and the diameter of *digraph*, the number of vertices at distance `i` from `u` and distance `j` from `v` depends only on `i`, `j`, and the distance between vertices `u` and `v`.

Alternatively, a distance regular graph is a graph for which there exist integers `b_i`, `c_i`, and `i` such that for any two vertices `u`, `v` in *digraph* which are distance `i` apart, there are exactly `b_i` neighbors of `v` which are at distance `i - 1` away from `u`, and `c_i` neighbors of `v` which are at distance `i + 1` away from `u`. This definition is used to check whether *digraph* is distance regular.

In the case where *digraph* is not symmetric or not connected, the property is `false`.

Example

```
gap> gr := DigraphSymmetricClosure(ChainDigraph(5));;
gap> IsDistanceRegularDigraph(gr);
false
gap> gr := Digraph([[2, 3, 4], [1, 3, 4], [1, 2, 4], [1, 2, 3]]);
<digraph with 4 vertices, 12 edges>
gap> IsDistanceRegularDigraph(gr);
true
```

6.3 Connectivity and cycles

6.3.1 IsAcyclicDigraph

▷ `IsAcyclicDigraph(`*digraph*`)` (property)

Returns: `true` or `false`.

Returns `true` if the digraph *digraph* is acyclic and `false` if it is not. A digraph is *acyclic* if there are no cycles, i.e. if there are no directed walks which start and end at the same vertex.

The method used in this operation has complexity $O(m+n)$ where m is the number of edges (counting multiple edges as one) and n is the number of vertices in the digraph.

Example

```
gap> Petersen := Graph( SymmetricGroup(5), [ [ 1, 2 ] ], OnSets,
> function(x, y) return IsEmpty(Intersection(x, y)); end );;
gap> gr:=Digraph(Petersen);
<digraph with 10 vertices, 30 edges>
gap> IsAcyclicDigraph(gr);
false
gap> gr:=Digraph( [ [  ], [ 1 ], [ 1 ], [ 1 ], [ 3 ], [ 3 ],
> [ 4 ], [ 4 ], [ 5 ], [ 5 ], [ 5 ], [ 6 ], [ 6 ], [ 7 ], [ 7 ],
> [ 7 ], [ 8 ], [ 9 ], [ 9 ], [ 11 ], [ 11 ], [ 12 ], [ 12 ], [ 13 ],
> [ 14 ], [ 15 ], [ 15 ], [ 16 ], [ 16 ], [ 17 ], [ 17 ], [ 18 ],
> [ 18 ], [ 19 ], [ 20 ], [ 20 ], [ 21 ], [ 22 ], [ 22 ], [ 23 ],
> [ 23 ], [ 24 ], [ 28 ], [ 29 ], [ 30 ], [ 30 ], [ 31 ], [ 32 ],
> [ 32 ], [ 33 ], [ 34 ], [ 41 ], [ 46 ], [ 47 ], [ 51 ] ] );;
gap> IsAcyclicDigraph(gr);
true
```

6.3.2 IsConnectedDigraph

▷ IsConnectedDigraph(*digraph*) (property)

Returns: true or false.

This property is true if the digraph *digraph* is weakly connected and false if it is not. A digraph *digraph* is *weakly connected* if it is possible to travel from any vertex to any other vertex by traversing edges in either direction (possibly against the orientation of some of them).

The method used in this function has complexity $O(m)$ if the digraph's DigraphSource (5.2.4) attribute is set, otherwise it has complexity $O(m+n)$ (where m is the number of edges and n is the number of vertices of the digraph).

Example
```
gap> gr := Digraph( [ [ 2 ], [ 3 ], [  ] ] );;
gap> IsConnectedDigraph(gr);
true
gap> gr := Digraph( [ [ 1, 3 ], [ 4 ], [ 3 ], [  ] ] );;
gap> IsConnectedDigraph(gr);
false
```

6.3.3 IsStronglyConnectedDigraph

▷ IsStronglyConnectedDigraph(*digraph*) (property)

Returns: true or false.

This property is true if the digraph *digraph* is strongly connected and false if it is not. A digraph *digraph* is *strongly connected* if there is a directed path from every vertex to every other vertex.

The method used in this operation is based on Gabow's Algorithm [Gab00] and has complexity $O(m+n)$, where m is the number of edges (counting multiple edges as one) and n is the number of vertices in the digraph.

Example
```
gap> gr := CycleDigraph(250000);
<digraph with 250000 vertices, 250000 edges>
gap> IsStronglyConnectedDigraph(gr);
true
gap> gr:=DigraphRemoveEdges(gr, [ [ 250000, 1 ] ]);
<digraph with 250000 vertices, 249999 edges>
gap> IsStronglyConnectedDigraph(gr);
false
```

6.3.4 IsAperiodicDigraph

▷ IsAperiodicDigraph(*digraph*) (property)

Returns: true or false.

This property is true if the digraph *digraph* is aperiodic, i.e. if its DigraphPeriod (5.3.12) is equal to 1. Otherwise, the property is false.

Example
```
gap> gr := Digraph( [ [ 6 ], [ 1 ], [ 2 ], [ 3 ], [ 4, 4 ], [ 5 ] ] );
<multidigraph with 6 vertices, 7 edges>
gap> IsAperiodicDigraph(gr);
false
gap> gr := Digraph( [ [ 2 ], [ 3, 5 ], [ 4 ], [ 5 ], [ 1, 2 ] ] );
```

```
<digraph with 5 vertices, 7 edges>
gap> IsAperiodicDigraph(gr);
true
```

Chapter 7

Homomorphisms

7.1 Acting on digraphs

7.1.1 OnDigraphs (for a digraph and a perm)

▷ `OnDigraphs(digraph, perm)` (operation)
▷ `OnDigraphs(digraph, trans)` (operation)

Returns: A digraph.

If *digraph* is a digraph, and the second argument *perm* is a *permutation* of the vertices of *digraph*, then this operation returns a digraph constructed by relabelling the vertices of *digraph* according to *perm*. Note that for an automorphism `f` of a digraph, we have `OnDigraphs(digraph, f) = digraph`.

If the second argument is a *transformation* *trans* of the vertices of *digraph*, then this operation returns a digraph constructed by transforming the source and range of each edge according to *trans*, and then removing any multiple edges. If *trans* is indeed a permutation, then the result coincides with relabelling the vertices of *digraph* according to *trans*.

The `DigraphVertexLabels` (5.1.10) of *digraph* will not be retained in the returned digraph.

Example

```
gap> gr := Digraph( [ [ 3 ], [ 1, 3, 5 ], [ 1 ], [ 1, 2, 4 ],
> [ 2, 3, 5 ] ] );
<digraph with 5 vertices, 11 edges>
gap> new := OnDigraphs(gr, (1,2));
<digraph with 5 vertices, 11 edges>
gap> OutNeighbours(new);
[ [ 2, 3, 5 ], [ 3 ], [ 2 ], [ 2, 1, 4 ], [ 1, 3, 5 ] ]
gap> gr := Digraph( [ [ 2 ], [  ], [ 2 ] ] );
<digraph with 3 vertices, 2 edges>
gap> t := Transformation( [ 1, 2, 1 ] );;
gap> new := OnDigraphs(gr, t);
<digraph with 3 vertices, 1 edge>
gap> OutNeighbours(new);
[ [ 2 ], [  ], [  ] ]
gap> ForAll(DigraphEdges(gr),
>  e -> IsDigraphEdge(new, [e[1] ^ t, e[2] ^ t]));
true
```

7.1.2 OnMultiDigraphs

▷ `OnMultiDigraphs(digraph, pair)` (operation)
▷ `OnMultiDigraphs(digraph, perm1, perm2)` (operation)

Returns: A digraph.

If *digraph* is a digraph, and *pair* is a pair consisting of a permutation of the vertices and a permutation of the edges of *digraph*, then this operation returns a digraph constructed by relabelling the vertices and edges of *digraph* according to *perm[1]* and *perm[2]*, respectively.

In its second form, `OnMultiDigraphs` returns a digraph with vertices and edges permuted by *perm1* and *perm2*, respectively.

Note that `OnDigraphs(digraph, perm)=OnMultiDigraphs(digraph, [perm, ()])` where `perm` is a permutation of the vertices of *digraph*. If you are only interested in the action of a permutation on the vertices of a digraph, then you can use `OnDigraphs` instead of `OnMultiDigraphs`.

Example
```
gap> gr1 := Digraph( [ [ 3, 6, 3 ], [  ], [ 3 ], [ 9, 10 ], [ 9 ],
> [  ],  [ ], [ 10, 4, 10 ], [  ], [  ] ] );
<multidigraph with 10 vertices, 10 edges>
gap> p := DigraphCanonicalLabelling(gr1);
[ (1,9,5,3,10,6,4,7), (1,7,9,5,2,8,4,10,3,6) ]
gap> gr2 := OnMultiDigraphs(gr1, p);
<multidigraph with 10 vertices, 10 edges>
gap> OutNeighbours(gr2);
[ [  ], [  ], [ 5 ], [  ], [  ], [  ], [ 5, 6 ], [ 6, 7, 6 ],
  [ 10, 4, 10 ], [ 10 ] ]
```

7.2 Isomorphisms, and Canonical labellings

7.2.1 AutomorphismGroup (for a digraph)

▷ `AutomorphismGroup(digraph)` (attribute)
▷ `AutomorphismGroup(digraph, colors)` (operation)

Returns: A permutation group.

If *digraph* is a digraph without multiple edges, then this function returns the automorphism group of *digraph*, as a group of permutations on the vertices of *digraph*.

If the *colors* argument is specified, then the group will consist of only those automorphisms which respect the given colouring. The colouring *colors* can be in one of two forms:

- A list of positive integers of size the number of vertices of *digraph*, where *colors*`[i]` is the colour of vertex `i`.
- A list of lists, such that *colors*`[i]` is a list of all vertices with colour `i`.

The automorphism group is found using bliss by Tommi Junttila and Petteri Kaski.

Example
```
gap> johnson := Digraph([[2, 3, 4, 5], [1, 3, 4, 6],
> [1, 2, 5, 6], [1, 2, 5, 6], [1, 3, 4, 6],
> [2, 3, 4, 5]]);
<digraph with 6 vertices, 24 edges>
gap> AutomorphismGroup(johnson);
Group([ (3,4), (2,3)(4,5), (1,2)(5,6) ])
```

```
gap> Size(last);
48
gap> cycle := CycleDigraph(9);
<digraph with 9 vertices, 9 edges>
gap> a := AutomorphismGroup(cycle);;
gap> StructureDescription(a);
"C9"
gap> a := AutomorphismGroup(cycle, [1, 2, 3, 1, 2, 3, 1, 2, 3]);;
gap> StructureDescription(a);
"C3"
gap> a := AutomorphismGroup(cycle, [[1, 4, 7], [2, 5, 8], [3, 6, 9]]);;
gap> StructureDescription(a);
"C3"
```

7.2.2 AutomorphismGroup (for a multidigraph)

▷ AutomorphismGroup(*digraph*) (attribute)

Returns: A direct product of permutation groups.

If *digraph* is a multidigraph, then this function returns the automorphism group of *digraph*, as a group of permutations on the vertices and edges of *digraph*.

For convenience, the returned group is the direct product of the group of automorphisms of the vertices of *digraph* with the stabiliser of the vertices in the automorphism group of the edges. These two groups can be accessed using the Projection (**Reference: Projection**) with second argument 1 and 2, respectively.

The permutations in the automorphism group of the edges act on the indices of the edges of *digraph*.

The automorphism group is found using bliss by Tommi Junttila and Petteri Kaski.

Example

```
gap> gr := DigraphEdgeUnion(CycleDigraph(3), CycleDigraph(3));
<multidigraph with 3 vertices, 6 edges>
gap> G := AutomorphismGroup(gr);
Group([ (1,2,3), (8,9), (6,7), (4,5) ])
gap> Range(Projection(G, 1));
Group([ (1,2,3) ])
gap> Range(Projection(G, 2));
Group([ (5,6), (3,4), (1,2) ])
gap> Size(G);
24
```

7.2.3 DigraphGroup

▷ DigraphGroup(*digraph*) (attribute)

Returns: A permutation group.

If *digraph* was created knowing a subgroup of its automorphism group, then this group is stored in the attribute DigraphGroup. If *digraph* is not created knowing a subgroup of it automorphism group, then DigraphGroup returns the entire automorphism group of *digraph*.

Note that certain other constructor operations such as CayleyDigraph (3.5.6), BipartiteDoubleDigraph (3.3.30), and DoubleDigraph (3.3.29), may not require a group as one of the arguments, but use the standard constructor method using a group, and hence set the DigraphGroup attribute for the resulting digraph.

Example

```
gap> n := 4;;
gap> adj := function(x, y)
>      return (((x-y) mod n) = 1) or (((x-y) mod n) = n-1);
>    end;;
gap> group := CyclicGroup(IsPermGroup, n);
Group([ (1,2,3,4) ])
gap> digraph := Digraph(group, [1..n], \^, adj);
<digraph with 4 vertices, 8 edges>
gap> HasDigraphGroup(digraph);
true
gap> DigraphGroup(digraph);
Group([ (1,2,3,4) ])
gap> AutomorphismGroup(digraph);
Group([ (2,4), (1,2,3,4) ])
gap> ddigraph := DoubleDigraph(digraph);
<digraph with 8 vertices, 32 edges>
gap> HasDigraphGroup(ddigraph);
true
gap> DigraphGroup(ddigraph);
Group([ (1,2,3,4)(5,6,7,8), (1,5)(2,6)(3,7)(4,8) ])
gap> AutomorphismGroup(ddigraph);
Group([ (6,8), (5,7), (4,6), (3,5), (2,4), (1,2,3,4)(5,6)(7,8) ])
gap> digraph := Digraph([[2, 3], [], []]);
<digraph with 3 vertices, 2 edges>
gap> HasDigraphGroup(digraph);
false
gap> HasAutomorphismGroup(digraph);
false
gap> DigraphGroup(digraph);
Group([ (2,3) ])
gap> HasAutomorphismGroup(digraph);
true
gap> group := DihedralGroup(8);
<pc group of size 8 with 3 generators>
gap> digraph := CayleyDigraph(group);
<digraph with 8 vertices, 24 edges>
gap> HasDigraphGroup(digraph);
true
gap> DigraphGroup(digraph);
Group([ (1,2)(3,8)(4,6)(5,7), (1,3,4,7)(2,5,6,8), (1,4)(2,6)(3,7)
(5,8) ])
```

7.2.4 DigraphSchreierVector

▷ DigraphSchreierVector(*digraph*) (attribute)

Returns: A list of integers.

DigraphSchreierVector returns the so-called *Schreier vector* of the action of the DigraphGroup (7.2.3) on the set of vertices of *digraph*. The Schreier vector is a list sch of integers with with length DigraphNrVertices(*digraph*) where:

`sch[i] < 0`:
implies that `i` is an orbit representative and `DigraphOrbitReps(`*digraph*`)[-sch[i]] = i`.

`sch[i] > 0`:
implies that `i / gens[sch[i]]` is one step closer to the root (or representative) of the tree, where gens is the generators of `DigraphGroup(`*digraph*`)`.

Example
```
gap> digraph := CayleyDigraph(AlternatingGroup(4));
<digraph with 12 vertices, 24 edges>
gap> sch := DigraphSchreierVector(digraph);
[ -1, 2, 2, 1, 1, 1, 1, 1, 2, 2, 2, 1 ]
gap> DigraphOrbitReps(digraph);
[ 1 ]
gap> gens := GeneratorsOfGroup(DigraphGroup(digraph));
[ (1,5,7)(2,4,8)(3,6,9)(10,11,12), (1,2,3)(4,7,10)(5,9,11)(6,8,12) ]
gap> 10 / gens[sch[10]];
7
gap> 7 / gens[sch[7]];
5
gap> 5 / gens[sch[5]];
1
```

7.2.5 DigraphOrbitReps

▷ `DigraphOrbitReps(`*digraph*`)` (attribute)

Returns: A list of integers.

`DigraphOrbitReps` returns a list of orbit representatives of the action of the `DigraphGroup` (7.2.3) on the set of vertices of *digraph*.

Example
```
gap> digraph := CayleyDigraph(AlternatingGroup(4));
<digraph with 12 vertices, 24 edges>
gap> DigraphOrbitReps(digraph);
[ 1 ]
gap> digraph := Digraph([[3, 8], [], [6, 8], [7], [2, 9], [10], [7], [1],
>                        [2, 7, 8], [8]]);
<digraph with 10 vertices, 14 edges>
gap> DigraphOrbitReps(digraph);
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

7.2.6 DigraphStabilizer

▷ `DigraphStabilizer(`*digraph, v*`)` (operation)

Returns: A permutation group.

`DigraphStabilizer` returns the stabilizer of the vertex *v* under of the action of the `DigraphGroup` (7.2.3) on the set of vertices of *digraph*.

Example
```
gap> digraph := Digraph([[2, 3, 5], [1, 4, 6], [4, 2, 7], [3, 1, 8], [6, 8, 1],
>                        [5, 7, 2], [8, 5, 3], [7, 6, 4]]);
<digraph with 8 vertices, 24 edges>
gap> DigraphStabilizer(digraph, 8);
```

```
Group(())
gap> DigraphStabilizer(digraph, 2);
Group(())
```

7.2.7 DigraphOrbits

▷ `DigraphOrbits(digraph)` (attribute)

Returns: A list of lists of integers.

`DigraphOrbits` returns the orbits of the action of the `DigraphGroup` (7.2.3) on the set of vertices of *digraph*.

Example

```
gap> G := Group((1,2,3), (1,2), (4,5,6), (7,8,9), (7,8));;
gap> gr := EdgeOrbitsDigraph(G, [1, 2]);
<digraph with 9 vertices, 6 edges>
gap> DigraphOrbits(gr);
[ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
```

7.2.8 RepresentativeOutNeighbours

▷ `RepresentativeOutNeighbours(digraph)` (attribute)

Returns: An immutable list of immutable lists.

This function returns the list `out` of *out-neighbours* of each representative of the orbits of the action of `DigraphGroup` (7.2.3) on the vertex set of the digraph *digraph*.

More specifically, if `reps` is the list of orbit representatives, then a vertex `j` appears in `out[i]` each time there exists an edge with source `reps[i]` and range `j` in *digraph*.

If `DigraphGroup` (7.2.3) is trivial, then `OutNeighbours` (5.2.5) is returned.

Example

```
gap> digraph := Digraph([[1, 2, 3, 4, 5], [3, 5], [2], [1, 2, 3, 5],
>                        [1, 2, 3, 4]]);
<digraph with 5 vertices, 16 edges>
gap> DigraphGroup(digraph);
Group(())
gap> RepresentativeOutNeighbours(digraph);
[ [ 1, 2, 3, 4, 5 ], [ 3, 5 ], [ 2 ], [ 1, 2, 3, 5 ], [ 1, 2, 3, 4 ] ]
gap> digraph := Digraph([[2, 3, 5], [1, 4, 6], [4, 2, 7], [3, 1, 8], [6, 8, 1],
>                        [5, 7, 2], [8, 5, 3], [7, 6, 4]]);
<digraph with 8 vertices, 24 edges>
gap> DigraphGroup(digraph);
Group([ (1,2)(3,4)(5,6)(7,8), (1,3,2,4)(5,7,6,8), (1,5)(2,6)(3,8)
(4,7) ])
gap> RepresentativeOutNeighbours(digraph);
[ [ 2, 3, 5 ] ]
```

7.2.9 DigraphCanonicalLabelling (for a digraph)

▷ `DigraphCanonicalLabelling(digraph)` (attribute)

▷ `DigraphCanonicalLabelling(digraph, colors)` (operation)

Returns: A permutation.

A function ρ from a digraph to a digraph is a *canonical representative map* if the following two conditions hold:

- $\rho(G)$ and G are isomorphic; and
- $\rho(G) = \rho(H)$ if and only if G and H are isomorphic.

A canonical labelling of a digraph *digraph* (under ρ) is an isomorphism of *digraph* onto its canonical representative $\rho($*digraph*$)$ given as a permutation of the vertices (and the edges in case *digraph* has multiple edges).

If the *colors* argument is specified, then the canonical labelling will respect the given colouring. The colouring *colors* can be in one of two forms:

- A list of positive integers of size the number of vertices of *digraph*, where *colors*`[i]` is the colour of vertex `i`.
- A list of lists, such that *colors*`[i]` is a list of all vertices with colour `i`.

The canonical labelling is found using bliss by Tommi Junttila and Petteri Kaski.

Example
```
gap> G := Digraph(10, [1, 1, 3, 4, 4, 5, 8, 8], [6, 3, 3, 9, 10, 9, 4, 10]);
<digraph with 10 vertices, 8 edges>
gap> DigraphCanonicalLabelling(G);
(1,8,5,3,10,6,4,9,7)
gap> DigraphCanonicalLabelling(G, [[1 .. 5], [6 .. 10]]);
(1,3,5,2)(6,7)(8,9,10)
gap> DigraphCanonicalLabelling(G, [1, 1, 1, 1, 2, 3, 1, 3, 2, 1]);
(1,4,5,7)(3,6,9,8,10)
gap> p := (1,2,7,5)(3,9)(6,10,8);;
gap> H := Digraph(
>  10,
>  OnTuples([1, 1, 3, 4, 4, 5, 8, 8], p),
>  OnTuples([6, 3, 3, 9, 10, 9, 4, 10], p)
> );
<digraph with 10 vertices, 8 edges>
gap> G = H;
false
gap> OnDigraphs(G, DigraphCanonicalLabelling(G)) =
>    OnDigraphs(H, DigraphCanonicalLabelling(H));
true
gap> gr := Digraph([[7, 2, 8, 2], [7, 6], [9], [],
> [], [], [], [2], [6], [4]]);
<multidigraph with 10 vertices, 10 edges>
gap> DigraphCanonicalLabelling(gr);
[ (1,9,7,5)(2,10,3), (1,7,3,8,2,6,10,4,5,9) ]
```

7.2.10 IsIsomorphicDigraph (for digraphs)

▷ `IsIsomorphicDigraph(`*digraph1, digraph2*`)` (operation)

Returns: `true` or `false`.

This operation returns `true` if the digraph *digraph1* is isomorphic to the digraph *digraph2*.

This operation uses the canonical labelling of the digraphs found with bliss by Tommi Junttila and Petteri Kaski.

Example
```
gap> digraph1 := CycleDigraph(4);
<digraph with 4 vertices, 4 edges>
gap> digraph2 := CycleDigraph(5);
<digraph with 5 vertices, 5 edges>
gap> IsIsomorphicDigraph(digraph1, digraph2);
false
gap> digraph2 := DigraphReverse(digraph1);
<digraph with 4 vertices, 4 edges>
gap> IsIsomorphicDigraph(digraph1, digraph2);
true
gap> digraph1 := Digraph([[], [9, 8, 9], [7],
> [], [], [5, 3], [8], [6], [4, 5], []]);
<multidigraph with 10 vertices, 10 edges>
gap> digraph2 := Digraph([[], [4], [], [7], [],
> [4, 5, 10, 5], [], [9], [4, 2], [2]]);
<multidigraph with 10 vertices, 10 edges>
gap> IsIsomorphicDigraph(digraph1, digraph2);
false
gap> digraph1 := Digraph([[3], [], []]);
<digraph with 3 vertices, 1 edge>
gap> digraph2 := Digraph([[], [], [2]]);
<digraph with 3 vertices, 1 edge>
gap> IsIsomorphicDigraph(digraph1, digraph2);
true
```

7.2.11 IsomorphismDigraphs (for digraphs)

▷ `IsomorphismDigraphs(`*digraph1*`, `*digraph2*`)` (operation)

Returns: A permutation or `fail`.

If *digraph1* and *digraph2* are isomorphic digraphs, then this operation returns an isomorphism from *digraph1* to *digraphs2*. More precisely,

for multidigraphs

this operation returns a pair of permutations `P` such that `OnMultiDigraphs(`*digraph1*`, P) = digraph2`. The first permutation is defined on the vertices of *digraph1* and the second on the edges.

for digraphs without multiple edges

this operation returns a permutation `p` such that `OnDigraphs(`*digraph1*`, p) = digraph2`.

If *digraph1* and *digraph2* are not isomorphic, then `fail` is returned.

This operation uses the canonical labelling of the digraphs found with bliss by Tommi Junttila and Petteri Kaski.

Example
```
gap> digraph1 := CycleDigraph(4);
<digraph with 4 vertices, 4 edges>
gap> digraph2 := CycleDigraph(5);
<digraph with 5 vertices, 5 edges>
gap> IsomorphismDigraphs(digraph1, digraph2);
fail
gap> gr1 := CompleteBipartiteDigraph(10, 5);
```

```
<digraph with 15 vertices, 100 edges>
gap> gr2 := CompleteBipartiteDigraph(5, 10);
<digraph with 15 vertices, 100 edges>
gap> p := IsomorphismDigraphs(gr1, gr2);
(1,6,11)(2,7,12)(3,8,13)(4,9,14)(5,10,15)
gap> OnDigraphs(gr1, p) = gr2;
true
gap> gr1 := Digraph([[3, 6, 3], [], [3], [9, 10], [9], [],
> [], [10, 4, 10], [], []]);
<multidigraph with 10 vertices, 10 edges>
gap> gr2 := Digraph([[], [], [5], [], [], [],
> [5, 6], [6, 7, 6], [10, 4, 10], [10]]);
<multidigraph with 10 vertices, 10 edges>
gap> IsomorphismDigraphs(gr1, gr2);
[ (1,9,5,3,10,6,4,7,2), (1,7)(2,8,4,10,6,3,9,5) ]
gap> digraph1 := Digraph([[], [], [], [], [], [],
> [10, 10], [], [], []]);
<multidigraph with 10 vertices, 2 edges>
gap> digraph2 := Digraph([[], [3, 3], [], [], [], [],
> [], [], [], []]);
<multidigraph with 10 vertices, 2 edges>
gap> IsomorphismDigraphs(digraph1, digraph2);
[ (2,4,6,8,9,10,3,5,7), () ]
```

7.3 Homomorphisms of digraphs

The following methods exist to find homomorphisms between digraphs. If an argument to one of these methods is a digraph with multiple edges, then the multiplicity of edges will be ignored in order to perform the calculation; the digraph will be treated as if it has no multiple edges.

7.3.1 HomomorphismDigraphsFinder

▷ HomomorphismDigraphsFinder(*gr1*, *gr2*, *hook*, *user_param*, *limit*, *hint*, *injective*, *image*, *map*) (function)

Returns: The argument *user_param*.

This function finds homomorphisms from the graph *gr1* to the graph *gr2* subject to the conditions imposed by the other arguments as described below.

If `f` and `g` are homomorphisms found by `HomomorphismGraphsFinder`, then `f` cannot be obtained from `g` by right multiplying by an automorphism of *gr2*.

hook

This argument should be a function or `fail`.

If *hook* is a function, then it should have two arguments *user_param* (see below) and a transformation `t`. The function *hook*(*user_param*, `t`) is called every time a new homomorphism `t` is found by `HomomorphismGraphsFinder`.

If *hook* is `fail`, then a default function is used which simply adds every new homomorphism found by `HomomorphismGraphsFinder` to *user_param*, which must be a list in this case.

user_param
: If *hook* is a function, then *user_param* can be any GAP object. The object *user_param* is used as the first argument for the function *hook*. For example, *user_param* might be a transformation semigroup, and *hook*(*user_param*, t) might set *user_param* to be the closure of *user_param* and t.

  If the value of *hook* is fail, then the value of *user_param* must be a list.

limit
: This argument should be a positive integer or infinity. HomomorphismGraphsFinder will return after it has found *limit* homomorphisms or the search is complete.

hint
: This argument should be a positive integer or fail.

  If *hint* is a positive integer, then only homorphisms of rank *hint* are found.

  If *hint* is fail, then no restriction is put on the rank of homomorphisms found.

injective
: This argument should be true or false. If it is true, then only injective homomorphisms are found, and if it is false there are no restrictions imposed by this argument.

image
: This argument should be a subset of the vertices of the graph *gr2*. HomomorphismGraphsFinder only finds homomorphisms from *gr1* to the subgraph of *gr2* induced by the vertices *image*.

map
: This argument should be a partial map from *gr1* to *gr2*, that is, a (not necessarily dense) list of vertices of the graph *gr2* of length no greater than the number vertices in the graph *gr1*. HomomorphismGraphsFinder only finds homomorphisms extending *map* (if any).

Example

```
gap> gr := ChainDigraph(10);
<digraph with 10 vertices, 9 edges>
gap> gr := DigraphSymmetricClosure(gr);
<digraph with 10 vertices, 18 edges>
gap> HomomorphismDigraphsFinder(gr, gr, fail, [], infinity, 2, false,
> [3, 4], [], fail, fail);
[ Transformation( [ 3, 4, 3, 4, 3, 4, 3, 4, 3, 4 ] ),
  Transformation( [ 4, 3, 4, 3, 4, 3, 4, 3, 4, 3 ] ) ]
gap> gr2 := CompleteDigraph(6);;
gap> HomomorphismDigraphsFinder(gr, gr2, fail, [], 1, fail, false,
> [1 .. 6], [1, 2, 1], fail, fail);
[ Transformation( [ 1, 2, 1, 3, 4, 5, 6, 1, 2, 1 ] ) ]
gap> func := function(user_param, t)
> Add(user_param, t * user_param[1]);
> end;;
gap> HomomorphismDigraphsFinder(gr, gr2, func, [Transformation([2, 2])],
> 3, fail, false, [1 .. 6], [1, 2, 1], fail, fail);
[ Transformation( [ 2, 2 ] ),
  Transformation( [ 2, 2, 2, 3, 4, 5, 6, 2, 2, 2 ] ),
  Transformation( [ 2, 2, 2, 3, 4, 5, 6, 2, 2, 3 ] ),
  Transformation( [ 2, 2, 2, 3, 4, 5, 6, 2, 2, 4 ] ) ]
```

7.3.2 DigraphHomomorphism

▷ `DigraphHomomorphism(digraph1, digraph2)` (operation)

Returns: A transformation, or `fail`.

A homomorphism from *digraph1* to *digraph2* is a mapping from the vertex set of *digraph1* to a subset of the vertices of *digraph2*, such that every pair of vertices `[i,j]` which has an edge `i->j` is mapped to a pair of vertices `[a,b]` which has an edge `a->b`. Note that non adjacent vertices can still be mapped onto adjacent ones.

`DigraphHomomorphism` returns a single homomorphism between *digraph1* and *digraph2* if it exists, otherwise it returns `fail`.

Example
```
gap> gr1 := ChainDigraph(3);;
gap> gr2 := Digraph([[3, 5], [2], [3, 1], [], [4]]);
<digraph with 5 vertices, 6 edges>
gap> DigraphHomomorphism(gr1, gr1);
IdentityTransformation
gap> DigraphHomomorphism(gr1, gr2);
Transformation( [ 1, 3, 1 ] )
```

7.3.3 HomomorphismsDigraphs

▷ `HomomorphismsDigraphs(digraph1, digraph2)` (operation)
▷ `HomomorphismsDigraphsRepresentatives(digraph1, digraph2)` (operation)

Returns: A list of transformations.

`HomomorphismsDigraphsRepresentatives` finds every `DigraphHomomorphism` (7.3.2) between *digraph1* and *digraph2*, up to right multiplication by an element of the `AutomorphismGroup` (7.2.1) of *digraph2*. In other words, every homomorphism `f` between *digraph1* and *digraph2* can be written as the composition `f = g * x`, where `g` is one of the `HomomorphismsDigraphsRepresentatives` and `x` is an automorphism of *digraph2*.

`HomomorphismsDigraphs` returns all homomorphisms between *digraph1* and *digraph2*.

Example
```
gap> gr1 := ChainDigraph(3);;
gap> gr2 := Digraph([[3, 5], [2], [3, 1], [], [4]]);
<digraph with 5 vertices, 6 edges>
gap> HomomorphismsDigraphs(gr1, gr2);
[ Transformation( [ 1, 3, 1 ] ), Transformation( [ 1, 3, 3 ] ),
  Transformation( [ 1, 5, 4, 4, 5 ] ), Transformation( [ 2, 2, 2 ] ),
  Transformation( [ 3, 1, 3 ] ), Transformation( [ 3, 1, 5, 4, 5 ] ),
  Transformation( [ 3, 3, 1 ] ), Transformation( [ 3, 3, 3 ] ) ]
gap> HomomorphismsDigraphsRepresentatives(gr1, CompleteDigraph(3));
[ IdentityTransformation, Transformation( [ 1, 2, 1 ] ) ]
```

7.3.4 DigraphMonomorphism

▷ `DigraphMonomorphism(digraph1, digraph2)` (operation)

Returns: A transformation, or `fail`.

`DigraphMonomorphism` returns a single *injective* `DigraphHomomorphism` (7.3.2) between *digraph1* and *digraph2* if one exists, otherwise it returns `fail`.

Example

```
gap> gr1 := ChainDigraph(3);;
gap> gr2 := Digraph([[3, 5], [2], [3, 1], [], [4]]);
<digraph with 5 vertices, 6 edges>
gap> DigraphMonomorphism(gr1, gr1);
IdentityTransformation
gap> DigraphMonomorphism(gr1, gr2);
Transformation( [ 1, 5, 4, 4, 5 ] )
```

7.3.5 MonomorphismsDigraphs

▷ MonomorphismsDigraphs(*digraph1, digraph2*) (operation)

▷ MonomorphismsDigraphsRepresentatives(*digraph1, digraph2*) (operation)

Returns: A list of transformations.

These operations behave the same as HomomorphismsDigraphs (7.3.3) and HomomorphismsDigraphsRepresentatives (7.3.3), expect they only return *injective* homomorphisms.

Example

```
gap> gr1 := ChainDigraph(3);;
gap> gr2 := Digraph([[3, 5], [2], [3, 1], [], [4]]);
<digraph with 5 vertices, 6 edges>
gap> MonomorphismsDigraphs(gr1, gr2);
[ Transformation( [ 1, 5, 4, 4, 5 ] ),
  Transformation( [ 3, 1, 5, 4, 5 ] ) ]
gap> MonomorphismsDigraphsRepresentatives(gr1, CompleteDigraph(3));
[ IdentityTransformation ]
```

7.3.6 DigraphEpimorphism

▷ DigraphEpimorphism(*digraph1, digraph2*) (operation)

Returns: A transformation, or fail.

DigraphEpimorphism returns a single *surjective* DigraphHomomorphism (7.3.2) between *digraph1* and *digraph2* if one exists, otherwise it returns fail.

Example

```
gap> gr1 := DigraphReverse(ChainDigraph(4));
<digraph with 4 vertices, 3 edges>
gap> gr2 := DigraphRemoveEdge(CompleteDigraph(3), [1, 2]);
<digraph with 3 vertices, 5 edges>
gap> DigraphEpimorphism(gr2, gr1);
fail
gap> DigraphEpimorphism(gr1, gr2);
Transformation( [ 1, 2, 3, 1 ] )
```

7.3.7 EpimorphismsDigraphs

▷ EpimorphismsDigraphs(*digraph1, digraph2*) (operation)

▷ EpimorphismsDigraphsRepresentatives(*digraph1, digraph2*) (operation)

Returns: A list of transformations.

These operations behave the same as `HomomorphismsDigraphs` (7.3.3) and `HomomorphismsDigraphsRepresentatives` (7.3.3), expect they only return *surjective* homomorphisms.

Example

```
gap> gr1 := DigraphReverse(ChainDigraph(4));
<digraph with 4 vertices, 3 edges>
gap> gr2 := DigraphSymmetricClosure(CycleDigraph(3));
<digraph with 3 vertices, 6 edges>
gap> EpimorphismsDigraphsRepresentatives(gr1, gr2);
[ Transformation( [ 1, 2, 3, 1 ] ), Transformation( [ 1, 2, 3, 2 ] ),
  Transformation( [ 1, 2, 1, 3 ] ) ]
gap> EpimorphismsDigraphs(gr1, gr2);
[ Transformation( [ 1, 2, 1, 3 ] ), Transformation( [ 1, 2, 3, 1 ] ),
  Transformation( [ 1, 2, 3, 2 ] ), Transformation( [ 1, 3, 1, 2 ] ),
  Transformation( [ 1, 3, 2, 1 ] ), Transformation( [ 1, 3, 2, 3 ] ),
  Transformation( [ 2, 1, 2, 3 ] ), Transformation( [ 2, 1, 3, 1 ] ),
  Transformation( [ 2, 1, 3, 2 ] ), Transformation( [ 2, 3, 1, 2 ] ),
  Transformation( [ 2, 3, 1, 3 ] ), Transformation( [ 2, 3, 2, 1 ] ),
  Transformation( [ 3, 1, 2, 1 ] ), Transformation( [ 3, 1, 2, 3 ] ),
  Transformation( [ 3, 1, 3, 2 ] ), Transformation( [ 3, 2, 1, 2 ] ),
  Transformation( [ 3, 2, 1, 3 ] ), Transformation( [ 3, 2, 3, 1 ] ) ]
```

7.3.8 GeneratorsOfEndomorphismMonoid

▷ `GeneratorsOfEndomorphismMonoid(digraph[, colors][, limit])` (function)

▷ `GeneratorsOfEndomorphismMonoidAttr(digraph)` (attribute)

Returns: A list of transformations.

An endomorphism of *digraph* is a homomorphism `DigraphHomomorphism` (7.3.2) from *digraph* back to itself. `GeneratorsOfEndomorphismMonoid`, called with a single argument, returns a generating set for the monoid of all endomorphisms of *digraph*.

If the *colors* argument is specified, then it will return a generating set for the monoid of endomorphisms which respect the given colouring. The colouring *colors* can be in one of two forms:

- A list of positive integers of size the number of vertices of *digraph*, where *colors*`[i]` is the colour of vertex `i`.
- A list of lists, such that *colors*`[i]` is a list of all vertices with colour `i`.

If the *limit* argument is specified, then it will return only the first *limit* homomorphisms, where *limit* must be a positive integer or `infinity`.

Example

```
gap> gr := Digraph(List([1 .. 3], x -> [1 .. 3]));;
gap> GeneratorsOfEndomorphismMonoid(gr);
[ Transformation( [ 1, 3, 2 ] ), Transformation( [ 2, 1 ] ),
  IdentityTransformation, Transformation( [ 1, 2, 1 ] ),
  Transformation( [ 1, 2, 2 ] ), Transformation( [ 1, 1, 2 ] ),
  Transformation( [ 1, 1, 1 ] ) ]
gap> GeneratorsOfEndomorphismMonoid(gr, 3);
[ Transformation( [ 1, 3, 2 ] ), Transformation( [ 2, 1 ] ),
  IdentityTransformation ]
gap> gr := CompleteDigraph(3);;
```

```
gap> GeneratorsOfEndomorphismMonoid(gr);
[ Transformation( [ 1, 3, 2 ] ), Transformation( [ 2, 1 ] ),
  IdentityTransformation ]
gap> GeneratorsOfEndomorphismMonoid(gr, [1, 2, 2]);
[ Transformation( [ 1, 3, 2 ] ), IdentityTransformation ]
gap> GeneratorsOfEndomorphismMonoid(gr, [[1], [2, 3]]);
[ Transformation( [ 1, 3, 2 ] ), IdentityTransformation ]
```

7.3.9 DigraphColoring (for a digraph and number of colors)

▷ `DigraphColoring(`*`digraph, n`*`)` (operation)
▷ `DigraphColouring(`*`digraph, n`*`)` (operation)
▷ `DigraphColoring(`*`digraph`*`)` (operation)

Returns: A transformation, or `fail`.

A digraph coloring is a labeling of the vertices (using precisely *n* colors) in such a way that two adjacent vertices can not have the same label. Alternatively, it can be defined to be a `DigraphEpimorphism` (7.3.6) from *digraph* onto a complete digraph with *n* vertices.

`DigraphColoring` returns such an epimorphism if one exists, else it returns `fail`.

Note that a digraph has a 2-coloring if and only if it is bipartite, see `IsBipartiteDigraph` (6.1.3).

If the third form is used, a greedy algorithm is used to obtain a colouring, which possibly is not minimal.

Example
```
gap> DigraphColoring(CompleteDigraph(5), 4);
fail
gap> DigraphColoring(ChainDigraph(10), 1);
fail
gap> gr := ChainDigraph(10);;
gap> t := DigraphColoring(gr, 2);
Transformation( [ 1, 2, 1, 2, 1, 2, 1, 2, 1, 2 ] )
gap> ForAll(DigraphEdges(gr), e -> e[1] ^ t <> e[2] ^ t);
true
gap> DigraphColoring(gr);
Transformation( [ 1, 2, 1, 2, 1, 2, 1, 2, 1, 2 ] )
```

7.3.10 DigraphEmbedding

▷ `DigraphEmbedding(`*`digraph1, digraph2`*`)` (operation)

Returns: A transformation, or `fail`.

An embedding of a digraph *digraph1* into another digraph *digraph2* is a `DigraphMonomorphism` (7.3.4) from *digraph1* to *digraph2* which has the additional property that a pair of vertices `[i,j]` which have no edge `i->j` in *digraph1* are mapped to a pair of vertices `[a,b]` which have no edge `a->b` in *digraph2*.

In other words, an embedding `t` is an isomorphism from *digraph1* to the `InducedSubdigraph` (3.3.2) of *digraph2* on the image of `t`.

`DigraphEmbedding` returns a single embedding if one exists, otherwise it returns `fail`.

Example
```
gap> gr := ChainDigraph(3);
<digraph with 3 vertices, 2 edges>
gap> DigraphEmbedding(gr, CompleteDigraph(4));
```

```
fail
gap> DigraphEmbedding(gr, Digraph([[3], [1, 4], [1], [3]]));
Transformation( [ 2, 4, 3, 4 ] )
```

Chapter 8

Finding cliques and independent sets

In Digraphs, a *clique* of a digraph is a set of mutually adjacent vertices of the digraph, and an *independent set* is a set of mutually non-adjacent vertices of the digraph. A *maximal clique* is a clique which is not properly contained in another clique, and a *maximal independent set* is an independent set which is not properly contained in another independent set. Using this definition in Digraphs, cliques and independent sets are both permitted, but not required, to contain vertices at which there is a loop. Another name for a clique is a *complete subgraph*.

Digraphs provides extensive functionality for computing cliques and independent sets of a digraph, whether maximal or not. The fundamental algorithm used in most of the methods in Digraphs to calculate cliques and independent sets is a version of the Bron-Kerbosch algorithm. Calculating the cliques and independent sets of a digraph is a well-known and hard problem, and searching for cliques or independent sets in a digraph can be very length, even for a digraph with a small number of vertices. Digraphs uses several strategies to increase the performance of these calculations.

From the definition of cliques and independent sets, it follows that the presence of loops and multiple edges in a digraph is irrelevant to the existence of cliques and independent sets in the digraph. See `DigraphHasLoops` (6.1.1) and `IsMultiDigraph` (6.1.7) for more information about these properties. Therefore given a digraph *digraph*, the cliques and independent sets of *digraph* are equal to the cliques and indepedents sets of the digraph:

- `DigraphRemoveLoops(DigraphRemoveAllMultipleEdges(`*digraph*`))`.

See `DigraphRemoveLoops` (3.3.21) and `DigraphRemoveAllMultipleEdges` (3.3.22) for more information about these attributes. Furthermore, the cliques of this digraph are equal to the cliques of the digraph formed by removing any edge `[u,v]` for which the corresponding reverse edge `[v,u]` is not present. Therefore, overall, the cliques of *digraph* are equal to the cliques of the symmetric digraph:

- `MaximalSymmetricSubdigraphWithoutLoops(`*digraph*`)`.

See `MaximalSymmetricSubdigraphWithoutLoops` (3.3.4) for more information about this attribute. The `AutomorphismGroup` (7.2.1) of this symmetric digraph contains the automorphism group of *digraph* as a subgroup. By performing the search for maximal cliques with the help of this larger automorphism group to reduce the search space, the computation time may be reduced. The functions and attributes which return representatives of cliques of *digraph* will return orbit representatives of cliques under the action of the automorphism group of the *maximal symmetric subdigraph without loops* on sets of vertices.

The independent sets of a digraph are equal to the independent sets of the `DigraphSymmetricClosure` (3.3.8). Therefore, overall, the independent sets of *digraph* are equal to the independent sets of the symmetric digraph:

- `DigraphSymmetricClosure(DigraphRemoveLoops(DigraphRemoveAllMultipleEdges(` *digraph*`)))`.

Again, the automorphism group of this symmetric digraph contains the automorphism group of *digraph*. Since a search for independent sets is equal to a search for cliques in the `DigraphDual` (3.3.7), the methods used in Digraphs usually transform a search for independent sets into a search for cliques, as described above. The functions and attributes which return representatives of independent sets of *digraph* will return orbit representatives of independent sets under the action of the automorphism group of the *symmetric closure* of the digraph formed by removing loops and multiple edges.

Please note that in Digraphs, cliques and indepedent sets are not required to be maximal. Some authors use the word clique to mean *maximal* clique, and some authors use the phrase independent set to mean *maximal* independent set, but please note that Digraphs does not use this definition.

8.1 Finding cliques

8.1.1 IsClique

▷ `IsClique(`*digraph, l*`)` (operation)

▷ `IsMaximalClique(`*digraph, l*`)` (operation)

Returns: `true` or `false`.

If *digraph* is a digraph and *l* is a duplicate-free list of vertices of *digraph*, then `IsClique(`*digraph*`,`*l*`)` returns `true` if *l* is a *clique* of *digraph* and `false` if it is not. Similarly, `IsMaximalClique(`*digraph*`,`*l*`)` returns `true` if *l* is a *maximal clique* of *digraph* and `false` if it is not.

A *clique* of a digraph is a set of mutually adjacent vertices of the digraph. A *maximal clique* is a clique which is not properly contained in another clique. A clique is permitted, but not required, to contain vertices at which there is a loop.

Example

```
gap> gr := CompleteDigraph(4);;
gap> IsClique(gr, [1, 3, 2]);
true
gap> IsMaximalClique(gr, [1, 3, 2]);
false
gap> IsMaximalClique(gr, DigraphVertices(gr));
true
gap> gr := Digraph([[1, 2, 4, 4], [1, 3, 4], [2, 1], [1, 2]]);
<multidigraph with 4 vertices, 11 edges>
gap> IsClique(gr, [2, 3, 4]);
false
gap> IsMaximalClique(gr, [1, 2, 4]);
true
```

8.1.2 CliquesFinder

▷ `CliquesFinder(`*`digraph, hook, user_param, limit, include, exclude, max, size, reps`*`)` (function)

Returns: The argument *`user_param`*.

This function finds cliques of the digraph *`digraph`* subject to the conditions imposed by the other arguments as described below. Note that a clique is represented by a list of the vertices which it contains.

Let `G` denote the automorphism group of maximal symmetric subdigraph of *`digraph`* without loops (see `AutomorphismGroup` (7.2.1) and `MaximalSymmetricSubdigraphWithoutLoops` (3.3.4)).

`hook`

This argument should be a function or `fail`.

If *`hook`* is a function, then it should have two arguments *`user_param`* (see below) and a clique `c`. The function *`hook`*`(`*`user_param`*`, c)` is called every time a new clique `c` is found by `CliquesFinder`.

If *`hook`* is `fail`, then a default function is used which simply adds every new clique found by `CliquesFinder` to *`user_param`*, which must be a list in this case.

`user_param`

If *`hook`* is a function, then *`user_param`* can be any GAP object. The object *`user_param`* is used as the first argument for the function *`hook`*. For example, *`user_param`* might be a list, and *`hook`*`(`*`user_param`*`, c)` might add the size of the clique `c` to the list *`user_param`*.

If the value of *`hook`* is `fail`, then the value of *`user_param`* must be a list.

`limit`

This argument should be a positive integer or `infinity`. `CliquesFinder` will return after it has found *`limit`* cliques or the search is complete.

`include` **and** *`exclude`*

These arguments should each be a (possibly empty) duplicate-free list of vertices of *`digraph`* (i.e. positive integers less than the number of vertices of *`digraph`*).

`CliquesFinder` will only look for cliques containing all of the vertices in *`include`* and containing none of the vertices in *`exclude`*.

Note that the search may be much more efficient if each of these lists is invariant under the action of `G` on sets of vertices.

`max` This argument should be `true` or `false`. If *`max`* is true then `CliquesFinder` will only search for *maximal* cliques. If `max` is `false` then non-maximal cliques may be found.

`size`

This argument should be `fail` or a positive integer. If *`size`* is a positive integer then `CliquesFinder` will only search for cliques which contain precisely *`size`* vertices. If *`size`* is `fail` then cliques of any size may be found.

`reps`

This argument should be `true` or `false`.

If *reps* is true then the arguments *include* and *exclude* are each required to be invariant under the action of G on sets of vertices. In this case, CliquesFinder will only find representatives of the orbits of the desired cliques under the action of G. If *reps* is false then CliquesFinder may find distinct cliques which are in the same orbit of G.

For a digraph such that G is non-trivial, the search for clique representatives can be much more efficient than the search for all cliques.

This function uses a version of the Bron-Kerbosch algorithm.

Example

```
gap> gr := CompleteDigraph(5);
<digraph with 5 vertices, 20 edges>
gap> user_param := [];;
gap> f := function(a, b) # Calculate size of clique
>   AddSet(user_param, Size(b));
> end;
function( a, b ) ... end
gap> CliquesFinder(gr, f, user_param, infinity, [], [], false, fail, true);
[ 1, 2, 3, 4, 5 ]
gap> CliquesFinder(gr, fail, [], 5, [2, 4], [3], false, fail, false);
[ [ 2, 4 ], [ 1, 2, 4 ], [ 2, 4, 5 ], [ 1, 2, 4, 5 ] ]
gap> CliquesFinder(gr, fail, [], 2, [2, 4], [3], false, fail, false);
[ [ 2, 4 ], [ 1, 2, 4 ] ]
gap> CliquesFinder(gr, fail, [], infinity, [], [], true, 5, false);
[ [ 1, 2, 3, 4, 5 ] ]
gap> CliquesFinder(gr, fail, [], infinity, [1, 3], [], false, 3, false);
[ [ 1, 2, 3 ], [ 1, 3, 4 ], [ 1, 3, 5 ] ]
gap> CliquesFinder(gr, fail, [], infinity, [1, 3], [], true, 3, false);
[  ]
```

8.1.3 DigraphClique

▷ DigraphClique(*digraph*[, *include*[, *exclude*[, *size*]]]) (function)

▷ DigraphMaximalClique(*digraph*[, *include*[, *exclude*[, *size*]]]) (function)

Returns: A list of positive integers, or fail.

If *digraph* is a digraph, then these functions returns a clique of *digraph* if one exists which satisfies the arguments, else it returns fail. A clique is defined by the set of vertices which it contains; see IsClique (8.1.1) and IsMaximalClique (8.1.1).

The optional arguments *include* and *exclude* must each be a (possibly empty) duplicate-free list of vertices of *digraph*, and the optional argument *size* must be a positive integer. By default, *include* and *exclude* are empty. These functions will search for a clique of *digraph* which includes the vertices of *include* and which does not include any vertices in *exclude*; if the argument *size* is supplied, then additionally the clique will be required to contain precisely *size* vertices.

If *include* is not a clique, then these functions return fail. Otherwise, the functions behave in the following way, depending on the number of arguments:

One or two arguments

If one or two arguments are supplied, then DigraphClique and DigraphMaximalClique greedily enlarge the clique *include* until it can not continue. The result is guaranteed to be a maximal clique. This may or may not return an answer more quickly than using DigraphMaximalCliques (8.1.4). with a limit of 1.

Three arguments

If three arguments are supplied, then `DigraphClique` greedily enlarges the clique *include* until it can not continue, although this clique may not be maximal.

Given three arguments, `DigraphMaximalClique` returns the maximal clique returned by `DigraphMaximalCliques(`*digraph*`,` *include*`,` *exclude*`, 1)` if one exists, else `fail`.

Four arguments

If four arguments are supplied, then `DigraphClique` returns the clique returned by `DigraphCliques(`*digraph*`,` *include*`,` *exclude*`, 1,` *size*`)` if one exists, else `fail`. This clique may not be maximal.

Given four arguments, `DigraphMaximalClique` returns the maximal clique returned by `DigraphMaximalCliques(`*digraph*`,` *include*`,` *exclude*`, 1,` *size*`)` if one exists, else `fail`.

Example
```
gap> gr := Digraph([[2, 3, 4], [1, 3], [1, 2], [1, 5], []]);
<digraph with 5 vertices, 9 edges>
gap> IsSymmetricDigraph(gr);
false
gap> DigraphClique(gr);
[ 5 ]
gap> DigraphMaximalClique(gr);
[ 5 ]
gap> DigraphClique(gr, [1, 2]);
[ 1, 2, 3 ]
gap> DigraphMaximalClique(gr, [1, 3]);
[ 1, 3, 2 ]
gap> DigraphClique(gr, [1], [2]);
[ 1, 4 ]
gap> DigraphMaximalClique(gr, [1], [3, 4]);
fail
gap> DigraphClique(gr, [1, 5]);
fail
gap> DigraphClique(gr, [], [], 2);
[ 1, 2 ]
```

8.1.4 DigraphMaximalCliques

▷ `DigraphMaximalCliques(`*digraph*`[,` *include*`[,` *exclude*`[,` *limit*`[,` *size*`]]]])` (function)

▷ `DigraphMaximalCliquesReps(`*digraph*`[,` *include*`[,` *exclude*`[,` *limit*`[,` *size*`]]]])` (function)

▷ `DigraphCliques(`*digraph*`[,` *include*`[,` *exclude*`[,` *limit*`[,` *size*`]]]])` (function)

▷ `DigraphCliquesReps(`*digraph*`[,` *include*`[,` *exclude*`[,` *limit*`[,` *size*`]]]])` (function)

▷ `DigraphMaximalCliquesAttr(`*digraph*`)` (attribute)

▷ `DigraphMaximalCliquesRepsAttr(`*digraph*`)` (attribute)

Returns: A list of lists of positive integers.

If *digraph* is digraph, then these functions and attributes use `CliquesFinder` (8.1.2) to return cliques of *digraph*. A clique is defined by the set of vertices which it contains; see `IsClique` (8.1.1) and `IsMaximalClique` (8.1.1).

The optional arguments *include* and *exclude* must each be a (possibly empty) list of vertices of *digraph*, the optional argument *limit* must be either a positive integer or `infinity`, and the optional argument *size* must be a positive integer. If not specified, then *include* and *exclude* are empty lists, and *limit* is `infinity`.

The functions will return as many suitable cliques as possible, up to the number *limit*. These functions will find cliques which contain all of the vertices of *include* and which do not contain any of the vertices of *exclude*. The argument *size* restricts the search to those cliques which contain precisely *size* vertices. If the function or attribute has `Maximal` in its name, then only maximal cliques will be returned; otherwise non-maximal cliques may be returned.

Let `G` denote the automorphism group of maximal symmetric subdigraph of *digraph* without loops (see `AutomorphismGroup` (7.2.1) and `MaximalSymmetricSubdigraphWithoutLoops` (3.3.4)).

Distinct cliques

`DigraphMaximalCliques` and `DigraphCliques` each return a duplicate-free list of at most *limit* cliques of *digraph* which satisfy the arguments.

The computation may be significantly faster if *include* and *exclude* are invariant under the action of `G` on sets of vertices.

Representatives of distinct orbits of cliques

To use `DigraphMaximalCliquesReps` or `DigraphCliquesReps`, the arguments *include* and *exclude* must each be invariant under the action of `G` on sets of vertices.

If this is the case, then `DigraphMaximalCliquesReps` and `DigraphCliquesReps` each return a duplicate-free list of at most *limit* orbits representatives (under the action of `G` on sets vertices) of cliques of *digraph* which satisfy the arguments.

Example

```
gap> gr := Digraph([[2, 3], [1, 3], [1, 2, 4], [3, 5, 6],
> [4, 6], [4, 5]]);
<digraph with 6 vertices, 14 edges>
gap> IsSymmetricDigraph(gr);
true
gap> G := AutomorphismGroup(gr);
Group([ (5,6), (1,2), (1,5)(2,6)(3,4) ])
gap> DigraphMaximalCliques(gr);
[ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 3, 4 ] ]
gap> DigraphMaximalCliquesReps(gr);
[ [ 1, 2, 3 ], [ 3, 4 ] ]
gap> Orbit(G, [1, 2, 3], OnSets);
[ [ 1, 2, 3 ], [ 4, 5, 6 ] ]
gap> Orbit(G, [3, 4], OnSets);
[ [ 3, 4 ] ]
gap> DigraphMaximalCliquesReps(gr, [3, 4], [], 1);
[ [ 3, 4 ] ]
gap> DigraphMaximalCliques(gr, [1, 2], [5, 6], 1, 2);
[  ]
gap> DigraphCliques(gr, [1], [5, 6], infinity, 2);
[ [ 1, 2 ], [ 1, 3 ] ]
```

8.2 Finding independent sets

8.2.1 IsIndependentSet

▷ `IsIndependentSet(`*`digraph, l`*`)` (operation)
▷ `IsMaximalIndependentSet(`*`digraph, l`*`)` (operation)

Returns: `true` or `false`.

If *digraph* is a digraph and *l* is a duplicate-free list of vertices of *digraph*, then `IsIndependentSet(`*`digraph`*`,`*`l`*`)` returns `true` if *l* is an *independent set* of *digraph* and `false` if it is not. Similarly, `IsMaximalIndependentSet(`*`digraph`*`,`*`l`*`)` returns `true` if *l* is a *maximal independent set* of *digraph* and `false` if it is not.

An *independent set* of a digraph is a set of mutually non-adjacent vertices of the digraph. A *maximal independent set* is an independent set which is not properly contained in another independent set. An independent set is permitted, but not required, to contain vertices at which there is a loop.

Example
```
gap> gr := CycleDigraph(4);;
gap> IsIndependentSet(gr, [1]);
true
gap> IsMaximalIndependentSet(gr, [1]);
false
gap> IsIndependentSet(gr, [1, 4, 3]);
false
gap> IsIndependentSet(gr, [2, 4]);
true
gap> IsMaximalIndependentSet(gr, [2, 4]);
true
```

8.2.2 DigraphIndependentSet

▷ `DigraphIndependentSet(`*`digraph[, include[, exclude[, size]]]`*`)` (function)
▷ `DigraphMaximalIndependentSet(`*`digraph[, include[, exclude[, size]]]`*`)` (function)

Returns: A lists of positive integers, or `fail`.

If *digraph* is a digraph, then these functions returns a independent set of *digraph* if one exists which satisfies the arguments, else it returns `fail`. A independent set is defined by the set of vertices which it contains; see `IsIndependentSet` (8.2.1) and `IsMaximalIndependentSet` (8.2.1).

The optional arguments *include* and *exclude* must each be a (possibly empty) duplicate-free list of vertices of *digraph*, and the optional argument *size* must be a positive integer. By default, *include* and *exclude* are empty. These functions will search for a independent set of *digraph* which includes the vertices of *include* and which does not include any vertices in *exclude*; if the argument *size* is supplied, then additionally the independent set will be required to contain precisely *size* vertices.

If *include* is not a independent set, then these functions return `fail`. Otherwise, the functions behave in the following way, depending on the number of arguments:

One or two arguments

If one or two arguments are supplied, then `DigraphIndependentSet` and `DigraphMaximalIndependentSet` greedily enlarge the independent set *include* until it can not continue. The result is guaranteed to be a maximal independent set. This may or may not return an answer more quickly than using `DigraphMaximalIndependentSets` (8.2.3). with a limit of 1.

Three arguments

If three arguments are supplied, then `DigraphIndependentSet` greedily enlarges the independent set *include* until it can not continue, although this independent set may not be maximal.

Given three arguments, `DigraphMaximalIndependentSet` returns the maximal independent set returned by `DigraphMaximalIndependentSets(`*digraph*`,` *include*`,` *exclude*`, 1)` if one exists, else `fail`.

Four arguments

If four arguments are supplied, then `DigraphIndependentSet` returns the independent set returned by `DigraphIndependentSets(`*digraph*`,` *include*`,` *exclude*`, 1,` *size*`)` if one exists, else `fail`. This independent set may not be maximal.

Given four arguments, `DigraphMaximalIndependentSet` returns the maximal independent set returned by `DigraphMaximalIndependentSets(`*digraph*`,` *include*`,` *exclude*`, 1,` *size*`)` if one exists, else `fail`.

Example
```
gap> gr := ChainDigraph(6);
<digraph with 6 vertices, 5 edges>
gap> DigraphIndependentSet(gr);
[ 6, 4, 2 ]
gap> DigraphMaximalIndependentSet(gr);
[ 6, 4, 2 ]
gap> DigraphIndependentSet(gr, [2, 4]);
[ 2, 4, 6 ]
gap> DigraphMaximalIndependentSet(gr, [1, 3]);
[ 1, 3, 6 ]
gap> DigraphIndependentSet(gr, [2, 4], [6]);
[ 2, 4 ]
gap> DigraphMaximalIndependentSet(gr, [2, 4], [6]);
fail
gap> DigraphIndependentSet(gr, [1], [], 2);
[ 1, 3 ]
gap> DigraphMaximalIndependentSet(gr, [1], [], 2);
fail
gap> DigraphMaximalIndependentSet(gr, [1], [], 3);
[ 1, 3, 5 ]
```

8.2.3 DigraphMaximalIndependentSets

▷ `DigraphMaximalIndependentSets(`*digraph*`[,` *include*`[,` *exclude*`[,` *limit*`[,` *size*`]]]])` (function)

▷ `DigraphMaximalIndependentSetsReps(`*digraph*`[,` *include*`[,` *exclude*`[,` *limit*`[,` *size*`]]]])` (function)

▷ `DigraphIndependentSets(`*digraph*`[,` *include*`[,` *exclude*`[,` *limit*`[,` *size*`]]]])` (function)

▷ `DigraphIndependentSetsReps(`*digraph*`[,` *include*`[,` *exclude*`[,` *limit*`[,` *size*`]]]])` (function)

▷ DigraphMaximalIndependentSetsAttr(*digraph*) (attribute)
▷ DigraphMaximalIndependentSetsRepsAttr(*digraph*) (attribute)

Returns: A list of lists of positive integers.

If *digraph* is digraph, then these functions and attributes use CliquesFinder (8.1.2) to return independent sets of *digraph*. An independent set is defined by the set of vertices which it contains; see IsMaximalIndependentSet (8.2.1) and IsIndependentSet (8.2.1).

The optional arguments *include* and *exclude* must each be a (possibly empty) list of vertices of *digraph*, the optional argument *limit* must be either a positive integer or infinity, and the optional argument *size* must be a positive integer. If not specified, then *include* and *exclude* are empty lists, and *limit* is infinity.

The functions will return as many suitable independent sets as possible, up to the number *limit*. These functions will find independent sets which contain all of the vertices of *include* and which do not contain any of the vertices of *exclude* The argument *size* restricts the search to those cliques which contain precisely *size* vertices. If the function or attribute has Maximal in its name, then only maximal independent sets will be returned; otherwise non-maximal independent sets may be returned.

Let G denote the AutomorphismGroup (7.2.1) of the DigraphSymmetricClosure (3.3.8) of the digraph formed from *digraph* by removing loops and ignoring the multiplicity of edges.

Distinct independent sets

DigraphMaximalIndependentSets and DigraphIndependentSets each return a duplicate-free list of at most *limit* independent sets of *digraph* which satisfy the arguments.

The computation may be significantly faster if *include* and *exclude* are invariant under the action of G on sets of vertices.

Representatives of distinct orbits of independent sets

To use DigraphMaximalIndependentSetsReps or DigraphIndependentSetsReps, the arguments *include* and *exclude* must each be invariant under the action of G on sets of vertices.

If this is the case, then DigraphMaximalIndependentSetsReps and DigraphIndependentSetsReps each return a duplicate-free list of at most *limit* orbits representatives (under the action of G on sets of vertices) of independent sets of *digraph* which satisfy the arguments.

Example

```
gap> gr := CycleDigraph(5);
<digraph with 5 vertices, 5 edges>
gap> DigraphMaximalIndependentSetsReps(gr);
[ [ 1, 3 ] ]
gap> DigraphIndependentSetsReps(gr);
[ [ 1 ], [ 1, 3 ] ]
gap> DigraphMaximalIndependentSets(gr);
[ [ 1, 3 ], [ 1, 4 ], [ 2, 4 ], [ 2, 5 ], [ 3, 5 ] ]
gap> DigraphMaximalIndependentSets(gr, [1]);
[ [ 1, 3 ], [ 1, 4 ] ]
gap> DigraphIndependentSets(gr, [], [4, 5]);
[ [ 1 ], [ 2 ], [ 3 ], [ 1, 3 ] ]
gap> DigraphIndependentSets(gr, [], [4, 5], 1, 2);
[ [ 1, 3 ] ]
```

Chapter 9

Visualising and IO

9.1 Visualising a digraph

9.1.1 Splash

▷ `Splash(str[, opts])` (function)

Returns: Nothing.

This function attempts to convert the string *`str`* into a pdf document and open this document, i.e. to splash it all over your monitor.

The string *`str`* must correspond to a valid `dot` or `LaTeX` text file and you must have have `GraphViz` and `pdflatex` installed on your computer. For details about these file formats, see `http://www.latex-project.org` and `http://www.graphviz.org`.

This function is provided to allow convenient, immediate viewing of the pictures produced by the function `DotDigraph` (9.1.2).

The optional second argument *`opts`* should be a record with components corresponding to various options, given below.

path this should be a string representing the path to the directory where you want `Splash` to do its work. The default value of this option is `"~/"`.

directory
this should be a string representing the name of the directory in `path` where you want `Splash` to do its work. This function will create this directory if does not already exist.

The default value of this option is `"tmp.viz"` if the option `path` is present, and the result of `DirectoryTemporary` (**Reference: DirectoryTemporary**) is used otherwise.

filename
this should be a string representing the name of the file where *`str`* will be written. The default value of this option is `"vizpicture"`.

viewer
this should be a string representing the name of the program which should open the files produced by `GraphViz` or `pdflatex`.

type this option can be used to specify that the string *`str`* contains a LaTeX or `dot` document. You can specify this option in *`str`* directly by making the first line `"%latex"` or `"//dot"`. There is no default value for this option, this option must be specified in *`str`* or in *`opt.type`*.

filetype

this should be a string representing the type of file which `Splash` should produce. For LaTeX files, this option is ignored and the default value `"pdf"` is used.

This function was written by Attila Egri-Nagy and Manuel Delgado with some minor changes by J. D. Mitchell.

Example

```
gap> Splash(DotDigraph(RandomDigraph(4)));
```

9.1.2 DotDigraph

▷ `DotDigraph(digraph)` (attribute)

Returns: A string.

This function produces a graphical representation of the digraph *digraph*. Vertices are displayed as circles, numbered consistently with *digraph*. Edges are displayed as arrowed lines between vertices, with the arrowhead of each line pointing towards the range of the edge.

The output is in `dot` format (also known as `GraphViz`) format. For details about this file format, and information about how to display or edit this format see `http://www.graphviz.org`.

The string returned by `DotDigraph` can be written to a file using the command `FileString` (**GAPDoc: FileString**).

Example

```
gap> adj := List( [ 1 .. 4 ], x -> List( [ 1 .. 4 ], y -> 1 ));
[ [ 1, 1, 1, 1 ], [ 1, 1, 1, 1 ], [ 1, 1, 1, 1 ], [ 1, 1, 1, 1 ] ]
gap> adj[1][1] := 1;
2
gap> adj[1][3] := 0;
0
gap> gr := DigraphByAdjacencyMatrix(adj);
<digraph with 4 vertices, 16 edges>
gap> FileString("dot/k4.dot", DotDigraph(gr));
146
```

9.1.3 DotSymmetricDigraph

▷ `DotSymmetricDigraph(digraph)` (attribute)

Returns: A string.

This function produces a graphical representation of the symmetric digraph *digraph*. `DotSymmetricDigraph` will return an error if *digraph* is not a symmetric digraph. See `IsSymmetricDigraph` (6.1.9).

Vertices are displayed as circles, numbered consistently with *digraph*. Since *digraph* is symmetric, for every non-loop edge there is a complementary edge with opposite source and range. `DotSymmetricDigraph` displays each pair of complementary edges as a single line between the relevant vertices, with no arrowhead.

The output is in `dot` format (also known as `GraphViz`) format. For details about this file format, and information about how to display or edit this format see `http://www.graphviz.org`.

The string returned by `DotSymmetricDigraph` can be written to a file using the command `FileString` (**GAPDoc: FileString**).

Example
```
gap> r := rec ( vertices := [ 1 .. 4 ],
> source := [ 1, 1, 1, 1, 2, 2, 3, 4, 4 ],
> range := [ 2, 2, 3, 4, 1, 1, 1, 1, 4 ] );;
gap> star := Digraph(r);
<digraph with 4 vertices, 9 edges>
gap> IsSymmetricDigraph(star);
true
gap> FileString("dot/star.dot", DotSymmetricDigraph(gr));;
```

9.2 Reading and writing graphs to a file

This section describes different ways to store and read graphs from a file in the Digraphs package.

Graph6
: *Graph6* is a graph data format for storing undirected graphs with no multiple edges nor loops of size up to $2^{36}-1$ in printable chracters. The format consists of two parts. The first part uses one to eight bytes to store the number of vertices. And the second part is the upper half of the adjacency matrix converted into ASCII characters. For a more detail description see Graph6.

Sparse6
: *Sparse6* is a graph data format for storing undirected graphs with possibly multiple edges or loops. The maximal number of vertices allowed is $2^{36}-1$. The format consists of two parts. The first part uses one to eight bytes to store the number of vertices. And the second part only stores information about the edges. Therefore, the *Sparse6* format return a more compact encoding then *Graph6* for sparse graph, i.e. graphs where the number of edges is much less than the number of vertices squared. For a more detail description see Sparse6.

Digraph6
: *Digraph6* is a new format based on *Graph6*, but designed for digraphs. The entire adjacency matrix is stored, and therefore there is support for directed edges and single-vertex loops. However, multiple edges are not supported.

DiSparse6
: *DiSparse6* is a new format based on *Sparse6*, but designed for digraphs. In this format the list of edges is partitioned into inceasing and decreasing edges, depending whether the edge has it's source bigger than the range. Then both sets of edges are written separetly in *Sparse6* format with a separation symbol in between.

9.2.1 DigraphFromGraph6String

▷ `DigraphFromGraph6String(str)` (operation)
▷ `DigraphFromDigraph6String(str)` (operation)
▷ `DigraphFromSparse6String(str)` (operation)
▷ `DigraphFromDiSparse6String(str)` (operation)

Returns: A digraph.

If *str* is a string encoding a graph in Graph6, Digraph6, Sparse6 or DiSparse6 format, then the corresponging function returns a digraph. In the case of either Graph6 or Sparse6, formats which do not support directed edges, this will be a digraph such that for every edge, the edge going in the opposite direction is also present.

Example

```
gap> DigraphFromGraph6String("?");
<digraph with 0 vertices, 0 edges>
gap> DigraphFromGraph6String("C]");
<digraph with 4 vertices, 8 edges>
gap> DigraphFromGraph6String("H?AAEM{");
<digraph with 9 vertices, 22 edges>
gap> DigraphFromDigraph6String("+?");
<digraph with 0 vertices, 0 edges>
gap> DigraphFromDigraph6String("+CQFG");
<digraph with 4 vertices, 6 edges>
gap> DigraphFromDigraph6String("+IM[SrKLc~lhesbU[F_");
<digraph with 10 vertices, 51 edges>
gap> DigraphFromDiSparse6String(".CaWBGA?b");
<multidigraph with 4 vertices, 9 edges>
```

9.2.2 Graph6String

▷ `Graph6String(`*`digraph`*`)` (operation)
▷ `Digraph6String(`*`digraph`*`)` (operation)
▷ `Sparse6String(`*`digraph`*`)` (operation)
▷ `DiSparse6String(`*`digraph`*`)` (operation)

Returns: A string.

These four functions return a highly compressed string fully describing the digraph *`digraph`*.

Graph6 and Digraph6 are formats best used on small, dense graphs, if applicable. For larger, sparse graphs use *Sparse6* and *Disparse6* (this latter also preserves multiple edges).

See `WriteDigraphs` (9.2.4).

Example

```
gap> gr := Digraph([[2,3], [1], [1]]);
<digraph with 3 vertices, 4 edges>
gap> Sparse6String(gr);
":Bc"
gap> DiSparse6String(gr);
".Bc{f"
```

9.2.3 ReadDigraphs

▷ `ReadDigraphs(`*`filename[, decoder][, n]`*`)` (function)

Returns: A digraph, or a list of digraphs.

If *`filename`* is the name of a file containing encoded digraphs (with one digraph per line), then `ReadDigraphs` returns these digraphs as a list. Note that if *`filename`* is a compressed file, which has been compressed appropriately to give a filename extension of `.gz`, `.bzip2`, or `.xz`, then `ReadDigraphs` can read *`filename`* without it first needing to be decompressed.

If the optional argument *`n`* is specified, then `ReadDigraphs` returns the digraph from the *`n`*th line of the file *`filename`*.

If the optional argument *`decoder`* is specified and is a function which decodes a string into a digraph, then `ReadDigraphs` will use *`decoder`* to decode the digraphs contained in *`filename`*. If the optional argument *decoder* is not specified, then `ReadDigraphs` will deduce which format to use based on the filename extension of *filename* (after removing the compression-related filename extensions `.gz`, `.bzip2`, and `.xz`). For example, if the filename extension if `.g6`, then `ReadDigraphs` will use the graph6 decoder.

The currently supported file formats, and associated filename extensions, are:

graph6 (.g6)
A standard and widely-used format for undirected graphs, with no support for loops or multiple edges. Only symmetric graphs are allowed – each edge is combined with its converse edge to produce a single undirected edge. This format is best used for "dense" graphs – those with many edges per vertex.

sparse6 (.s6)
Unlike graph6, sparse6 has support for loops and multiple edges. However, its use is still limited to symmetric graphs. This format is better-suited to "sparse" graphs – those with few edges per vertex.

digraph6 (.d6)
This format is based on graph6, but stores direction information - therefore is not limited to symmetric graphs. Loops are allowed, but multiple edges are not. Best compression with "dense" graphs.

disparse6 (.ds6)
Any type of digraph can be encoded in disparse6: directions, loops, and multiple edges are all allowed. Similar to sparse6, this has the best compression rate with "sparse" graphs.

plain text (.txt)
This is a human-readable format which stores graphs in the form `0 7 0 8 1 7 2 8 3 8 4 8 5 8 6 8` i.e. pairs of vertices describing edges in a graph. More specifically, the vertices making up one edge must be separated by a single space, and pairs of vertices must be separated by two spaces.

See `ReadPlainTextDigraph` (9.2.10) for a more flexible way to store digraphs in a plain text file.

pickled (`.p` or `.pickle`)
Digraphs are pickled using the IO package. This is particularly good when the `DigraphGroup` (7.2.3) is non-trivial.

Example

```
gap> ReadDigraphs(
> Concatenation(DIGRAPHS_Dir(), "/data/graph5.g6.gz"), 10);
<digraph with 5 vertices, 8 edges>
gap> ReadDigraphs(
> Concatenation(DIGRAPHS_Dir(), "/data/graph5.g6.gz"), 17);
<digraph with 5 vertices, 12 edges>
gap> ReadDigraphs(
> Concatenation(DIGRAPHS_Dir(), "/data/tree9.4.txt"));
[ <digraph with 9 vertices, 8 edges>,
  <digraph with 9 vertices, 8 edges>,
  <digraph with 9 vertices, 8 edges>,
  <digraph with 9 vertices, 8 edges>,
```

```
<digraph with 9 vertices, 8 edges>,
<digraph with 9 vertices, 8 edges>,
<digraph with 9 vertices, 8 edges>,
<digraph with 9 vertices, 8 edges>,
<digraph with 9 vertices, 8 edges>,
<digraph with 9 vertices, 8 edges>,
<digraph with 9 vertices, 8 edges>,
<digraph with 9 vertices, 8 edges>,
<digraph with 9 vertices, 8 edges>,
<digraph with 9 vertices, 8 edges> ]
```

9.2.4 WriteDigraphs

▷ `WriteDigraphs(`*`filename, digraphs`*`)` (function)

If *digraphs* is a list of digraphs, then `WriteDigraphs` writes its contents to a file in a compact format. If the supplied filename ends in one of the extensions `.gz`, `.bzip2`, or `.xz`, the file will be compressed appropriately. Excluding these extensions, if the file ends with an extension in the list below, the corresponding graph format will be used to encode it. If such an extension is not included, an appropriate format will be chosen intelligently, and an extension appended, to minimise file length.

For more verbose information on the progress of the function, set the info level of *InfoDigraphs* to 1 or higher, using `SetInfoLevel`.

The currently supported file formats are:

graph6 (.g6)
A standard and widely-used format for undirected graphs, with no support for loops or multiple edges. Only symmetric graphs are allowed – each edge is combined with its converse edge to produce a single undirected edge. This format is best used for "dense" graphs – those with many edges per vertex.

sparse6 (.s6)
Unlike graph6, sparse6 has support for loops and multiple edges. However, its use is still limited to symmetric graphs. This format is better-suited to "sparse" graphs – those with few edges per vertex.

digraph6 (.d6)
This format is based on graph6, but stores direction information - therefore is not limited to symmetric graphs. Loops are allowed, but multiple edges are not. Best compression with "dense" graphs.

disparse6 (.ds6)
Any type of digraph can be encoded in disparse6: directions, loops, and multiple edges are all allowed. Similar to sparse6, this has the best compression rate with "sparse" graphs.

plain text (.txt)
This is a human-readable format which stores graphs in the form `0 7  0 8  1 7  2 8  3 8  4 8  5 8  6 8` i.e. pairs of vertices describing edges in a graph. More specifically, the vertices making up one edge must be separated by a single space, and pairs of vertices must be separated by two spaces.

See `ReadPlainTextDigraph` (9.2.10) for a more flexible way to store digraphs in a plain text file.

pickled (`.p` **or** `.pickle`)
Digraphs are pickled using the IO package. This is particularly good when the `DigraphGroup` (7.2.3) is non-trivial.

Example
```
gap> grs := [];;
gap> grs[1] := Digraph([]);
<digraph with 0 vertices, 0 edges>
gap> grs[2] := Digraph([[1, 3], [2], [1, 2]]);
<digraph with 3 vertices, 5 edges>
gap> grs[3] := Digraph([[6, 7], [6, 9], [1, 3, 4, 5, 8, 9],
> [1, 2, 3, 4, 5, 6, 7, 10], [1, 5, 6, 7, 10], [2, 4, 5, 9, 10],
> [3, 4, 5, 6, 7, 8, 9, 10], [1, 3, 5, 7, 8, 9], [1, 2, 5],
> [1, 2, 4, 6, 7, 8]]);
<digraph with 10 vertices, 51 edges>
gap> filename := Concatenation(DIGRAPHS_Dir(), "/tst/out/man.d6.gz");;
gap> WriteDigraphs(filename, grs, "w");
gap> ReadDigraphs(filename);
[ <digraph with 0 vertices, 0 edges>,
  <digraph with 3 vertices, 5 edges>,
  <digraph with 10 vertices, 51 edges> ]
```

9.2.5 DigraphPlainTextLineEncoder

▷ `DigraphPlainTextLineEncoder(`*`delimiter1[, delimiter2], offset`*`)` (function)
▷ `DigraphPlainTextLineDecoder(`*`delimiter1[, delimiter2], offset`*`)` (function)

Returns: A string.

These two functions return a function which encodes or decodes a digraph in a plain text format.

`DigraphPlainTextLineEncoder` returns a function which takes a single digraph as an argument. The function returns a string describing the edges of that digraph; each edge is written as a pair of integers separated by the string *`delimiter2`*, and the edges themselves are separated by the string *`delimiter1`*. *`DigraphPlainTextLineDecoder`* returns the corresponding decoder function, which takes a string argument in this format and returns a digraph.

If only one delimiter is passed as an argument to *`DigraphPlainTextLineDecoder`*, it will return a function which decodes a single edge, returning its contents as a list of integers.

The argument *`offset`* should be an integer, which will describe a number to be added to each vertex before it is encoded, or after it is decoded. This may be used, for example, to label vertices starting at 0 instead of 1.

Note that the number of vertices of a digraph is not stored, and so vertices which are not connected to any edge may be lost.

Example
```
gap> gr := Digraph([[2,3], [1], [1]]);
<digraph with 3 vertices, 4 edges>
gap> enc := DigraphPlainTextLineEncoder("  ", " ", -1);;
gap> dec := DigraphPlainTextLineDecoder("  ", " ", 1);;
gap> enc(gr);
"0 1  0 2  1 0  2 0"
```

```
gap> dec(last);
<digraph with 3 vertices, 4 edges>
```

9.2.6 TournamentLineDecoder

▷ `TournamentLineDecoder(str)` (function)

Returns: A digraph.

This function takes a string *str*, decodes it, and then returns the tournament [see `IsTournament` (6.1.10)] which it defines, according to the following rules.

The characters of the string *str* represent the entries in the upper triangle of a tournament's adjacency matrix. The number of vertices `n` will be detected from the length of the string and will be as large as possible.

The first character represents the possible edge `1 -> 2`, the second represents `1 -> 3` and so on until `1 -> n`; then the following character represents `2 -> 3`, and so on up to the character which represents the edge `n-1 -> n`.

If a character of the string with corresponding edge `i -> j` is equal to `1`, then the edge `i -> j` is present in the tournament. Otherwise, the edge `i -> j` is present instead. In this way, all the possible edges are encoded one-by-one.

Example
```
gap> gr := TournamentLineDecoder("100001");
<digraph with 4 vertices, 6 edges>
gap> OutNeighbours(gr);
[ [ 2 ], [  ], [ 1, 2, 4 ], [ 1, 2 ] ]
```

9.2.7 AdjacencyMatrixUpperTriangleLineDecoder

▷ `AdjacencyMatrixUpperTriangleLineDecoder(str)` (function)

Returns: A digraph.

This function takes a string *str*, decodes it, and then returns the topologically sorted digraph [see `DigraphTopologicalSort` (5.1.7)] which it defines, according to the following rules.

The characters of the string *str* represent the entries in the upper triangle of a digraph's adjacency matrix. The number of vertices `n` will be detected from the length of the string and will be as large as possible.

The first character represents the possible edge `1 -> 2`, the second represents `1 -> 3` and so on until `1 -> n`; then the following character represents `2 -> 3`, and so on up to the character which represents the edge `n-1 -> n`. If a character of the string with corresponding edge `i -> j` is equal to `1`, then this edge is present in the digraph. Otherwise, it is not present. In this way, all the possible edges are encoded one-by-one.

In particular, note that there exists no edge `[i, j]` if $j \leq i$. In order words, the digraph will be topologically sorted.

Example
```
gap> gr := AdjacencyMatrixUpperTriangleLineDecoder("100001");
<digraph with 4 vertices, 2 edges>
gap> OutNeighbours(gr);
[ [ 2 ], [  ], [ 4 ], [  ] ]
gap> gr := AdjacencyMatrixUpperTriangleLineDecoder("111111x111");
<digraph with 5 vertices, 9 edges>
```

```
gap> OutNeighbours(gr);
[ [ 2, 3, 4, 5 ], [ 3, 4 ], [ 4, 5 ], [ 5 ], [  ] ]
```

9.2.8 TCodeDecoder

▷ `TCodeDecoder(`*`str`*`)` (function)

Returns: A digraph.

If *str* is a string consisting of at least two non-negative integers separated by spaces, then this function will attempt to return the digraph which it defines as a TCode string.

The first integer of the string defines the number of vertices `v` in the digraph, and the second defines the number of edges `e`. The following `2e` integers should be vertex numbers in the range `[0 .. v-1]`. These integers are read in pairs and define the digraph's edges. This function will return an error if *str* has fewer than `2e+2` entries.

Note that the vertex numbers will be incremented by 1 in the digraph returned. Hence the string fragment `0 6` will describe the edge `[1,7]`.

Example

```
gap> gr := TCodeDecoder("3 2 0 2 2 1");
<digraph with 3 vertices, 2 edges>
gap> OutNeighbours(gr);
[ [ 3 ], [  ], [ 2 ] ]
gap> gr := TCodeDecoder("12 3 0 10 5 2 8 8");
<digraph with 12 vertices, 3 edges>
gap> OutNeighbours(gr);
[ [ 11 ], [  ], [  ], [  ], [  ], [ 3 ], [  ], [  ], [ 9 ], [  ],
  [  ], [  ] ]
```

9.2.9 PlainTextString

▷ `PlainTextString(`*`digraph`*`)` (operation)

▷ `DigraphFromPlainTextString(`*`s`*`)` (operation)

Returns: A string.

`PlainTextString` takes a single digraph, and returns a string describing the edges of that digraph. *`DigraphFromPlainTextString`* takes such a string and returns the digraph which it describes. Each edge is written as a pair of integers separated by a single space. The edges themselves are separated by a double space. Vertex numbers are reduced by 1 when they are encoded, so that vertices in the string are labelled starting at 0.

Note that the number of vertices of a digraph is not stored, and so vertices which are not connected to any edge may be lost.

Example

```
gap> gr := Digraph([[2,3], [1], [1]]);
<digraph with 3 vertices, 4 edges>
gap> PlainTextString(gr);
"0 1  0 2  1 0  2 0"
gap> DigraphFromPlainTextString(last);
<digraph with 3 vertices, 4 edges>
```

9.2.10 WritePlainTextDigraph

▷ `WritePlainTextDigraph(filename, digraph, delimiter, offset)` (function)

▷ `ReadPlainTextDigraph(filename, delimiter, offset, ignore)` (function)

These functions write and read a single digraph in a human-readable plain text format as follows: each line contains a single edge, and each edge is written as a pair of integers separated by the string *delimiter*.

filename should be the name of a file which will be written to or read from, and *offset* should be an integer which is added to each vertex number as it is written or read. For example, if `WritePlainTextDigraph` is called with *offset* `-1`, then the vertices will be numbered in the file starting from 0 instead of 1 - `ReadPlainTextDigraph` would then need to be called with *offset* `1` to convert back to the original graph.

ignore should be a list of characters which will be ignored when reading the graph.

Example

```
gap> gr := Digraph([[1,2,3], [1,1], [2]]);
<multidigraph with 3 vertices, 6 edges>
gap> filename := Concatenation(DIGRAPHS_Dir(), "/tst/out/plain.txt");;
gap> WritePlainTextDigraph(filename, gr, ",", -1);
gap> ReadPlainTextDigraph(filename, ",", 1, ['/','%']);
<multidigraph with 3 vertices, 6 edges>
```

Appendix A

Grape to Digraphs Command Map

Below is a table of Grape commands with the Digraphs counterparts. The sections in this chapter correspond to the chapters in the Grape manual.

A.1 Functions to construct and modify graphs

The table in this section contains more information when viewed in html format.

Grape command	Digraphs command	
Graph	Digraph (3.1.5)	
EdgeOrbitsGraph	EdgeOrbitsDigraph (3.1.8)	
NullGraph	NullDigraph (3.5.5)	
CompleteGraph	CompleteDigraph (3.5.2)	
JohnsonGraph	JohnsonDigraph (3.5.7)	
CayleyGraph	CayleyDigraph (3.5.6)	
AddEdgeOrbit	DigraphAddEdgeOrbit (3.3.14)	
RemoveEdgeOrbit	DigraphRemoveEdgeOrbit (3.3.19)	
AssignVertexNames	SetDigraphVertexLabels (5.1.10)	

A.2 Functions to inspect graphs, vertices and edges

The table in this section contains more information when viewed in html format.

Grape command	Digraphs command	
`IsGraph`	`IsDigraph` (3.1.1)	
`OrderGraph`	`DigraphNrVertices` (5.1.2)	
`IsVertex(`*graph*`, `*v*`)`	*v* `in DigraphVertices(`*digraph*`)`	
`VertexName`	`DigraphVertexLabel` (5.1.9)	
`VertexNames`	`DigraphVertexLabels` (5.1.10)	
`Vertices`	`DigraphVertices` (5.1.1)	
`VertexDegree`	`OutDegreeOfVertex` (5.2.9)	
`VertexDegrees`	`OutDegreeSet` (5.2.7)	
`IsLoopy`	`DigraphHasLoops` (6.1.1)	
`IsSimpleGraph`	`IsSymmetricDigraph` (6.1.9)	
`Adjacency`	`OutNeighboursOfVertex` (5.2.10)	
`IsEdge`	`IsDigraphEdge` (5.1.13)	
`DirectedEdges`	`DigraphEdges` (5.1.3)	
`UndirectedEdges`	None	
`Distance`	`DigraphShortestDistance` (5.3.2)	
`Diameter`	`DigraphDiameter` (5.3.1)	
`Girth`	`DigraphUndirectedGirth` (5.3.7)	
`IsConnectedGraph`	`IsStronglyConnectedDigraph` (6.3.3)	
`IsBipartite`	`IsBipartiteDigraph` (6.1.3)	
`IsNullGraph`	`IsNullDigraph` (6.1.5)	
`IsCompleteGraph`	`IsCompleteDigraph` (6.1.4)	

A.3 Functions to determine regularity properties of graphs

The table in this section contains more information when viewed in html format.

Grape command	Digraphs command	
`IsRegularGraph`	`IsOutRegularDigraph` (6.2.2)	
`LocalParameters`	None	
`GlobalParameters`	None	
`IsDistanceRegular`	`IsDistanceRegularDigraph` (6.2.4)	
`CollapsedAdjacencyMat`	None	
`OrbitalGraphColadjMats`	None	
`VertexTransitiveDRGs`	None	

A.4 Some special vertex subsets of a graph

The table in this section contains more information when viewed in html format.

Grape command	Digraphs command	
`ConnectedComponent`	`DigraphConnectedComponent` (5.3.9)	
`ConnectedComponents`	`DigraphConnectedComponents` (5.3.8)	
`Bicomponents`	`DigraphBicomponents` (5.1.8)	
`DistanceSet`	`DigraphDistanceSet` (5.3.5)	
`Layers`	`DigraphLayers` (5.3.17)	
`IndependentSet`	`DigraphIndependentSet` (8.2.2)	

A.5 Functions to construct new graphs from old

The table in this section contains more information when viewed in html format.

Grape command	Digraphs command	
InducedSubgraph	InducedSubdigraph (3.3.2)	
DistanceSetInduced	None	
DistanceGraph	DistanceDigraph (3.3.32)	
ComplementGraph	DigraphDual (3.3.7)	
PointGraph	None	
EdgeGraph	EdgeUndirectedDigraph (3.3.28)	
SwitchedGraph	None	
UnderlyingGraph	DigraphSymmetricClosure (3.3.8)	
QuotientGraph	QuotientDigraph (3.3.5)	
BipartiteDouble	BipartiteDoubleDigraph (3.3.30)	
GeodesicsGraph	None	
CollapsedIndependentOrbitsGraph	None	
CollapsedCompleteOrbitsGraph	None	
NewGroupGraph	None	

A.6 Vertex-Colouring and Complete Subgraphs

The table in this section contains more information when viewed in html format.

Grape command	Digraphs command	
VertexColouring	DigraphColoring (7.3.9)	
CompleteSubgraphs	DigraphCliques (8.1.4)	
CompleteSubgraphsOfGivenSize	DigraphCliques (8.1.4)	

A.7 Automorphism groups and isomorphism testing for graphs

The table in this section contains more information when viewed in html format.

Grape command	Digraphs command	
AutGroupGraph	AutomorphismGroup (7.2.1)	
GraphIsomorphism	IsomorphismDigraphs (7.2.11)	
IsIsomorphicGraph	IsIsomorphicDigraph (7.2.10)	
GraphIsomorphismClassRepresentatives	None	

References

[CK86] R. Calderbank and W. M. Kantor. The geometry of two-weight codes. *Bull. London Math. Soc.*, 18(2):97–122, 1986. 11

[Gab00] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(34):107 – 114, 2000. 51, 63

[JK07] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In David Applegate, Gerth Stølting Brodal, Daniel Panario, and Robert Sedgewick, editors, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*, pages 135–149. SIAM, 2007. 5

[vLS81] J. H. van Lint and A. Schrijver. Construction of strongly regular graphs, two-weight codes and partial geometries by finite fields. *Combinatorica*, 1(1):63–73, 1981. 11

Index