Digraphs
Version 0.15.2

Jan De Beule
Julius Jonušas
James D. Mitchell
Michael Torpey
Wilf A. Wilson
Stuart Burrell
Luke Elliott
Christopher Jefferson
Markus Pfeiffer
Chris Russell
Finn Smith

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

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

James D. Mitchell  Email: jdm3@st-andrews.ac.uk
Homepage: http://goo.gl/ZtViV6

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

Wilf A. Wilson  Email: gap@wilf-wilson.net
Homepage: http://wilf.me
Abstract

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

Copyright


Digraphs is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.

Acknowledgements

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

1 The Digraphs package ................................................................................................. 5
   1.1 Introduction ........................................................................................................ 5

2 Installing Digraphs .................................................................................................. 7
   2.1 For those in a hurry .......................................................................................... 7
   2.2 Optional package dependencies ...................................................................... 8
   2.3 Compiling the kernel module .......................................................................... 8
   2.4 Rebuilding the documentation ....................................................................... 9
   2.5 Testing your installation ............................................................................... 9

3 Creating digraphs .................................................................................................... 10
   3.1 Creating digraphs ......................................................................................... 10
   3.2 Changing representations .............................................................................. 15
   3.3 New digraphs from old .................................................................................. 17
   3.4 Random digraphs ......................................................................................... 33
   3.5 Standard examples ....................................................................................... 34

4 Operators .................................................................................................................. 37
   4.1 Operators for digraphs .................................................................................. 37

5 Attributes and operations ......................................................................................... 40
   5.1 Vertices and edges ......................................................................................... 40
   5.2 Neighbours and degree .................................................................................. 47
   5.3 Reachability and connectivity ...................................................................... 54
   5.4 Cayley graphs of groups ............................................................................... 67
   5.5 Associated semigroups .................................................................................. 68
   5.6 Planarity .......................................................................................................... 69

6 Properties of digraphs .............................................................................................. 73
   6.1 Edge properties .............................................................................................. 73
   6.2 Regularity ......................................................................................................... 80
   6.3 Connectivity and cycles ................................................................................ 82
   6.4 Planarity .......................................................................................................... 87

7 Homomorphisms ....................................................................................................... 89
   7.1 Acting on digraphs ......................................................................................... 89
   7.2 Isomorphisms and canonical labellings ......................................................... 90
7.3 Homomorphisms of digraphs .................................................. 106

8 Finding cliques and independent sets ................................. 116
  8.1 Finding cliques ................................................................. 117
  8.2 Finding independent sets ................................................... 122

9 Visualising and IO ................................................................. 126
  9.1 Visualising a digraph .......................................................... 126
  9.2 Reading and writing graphs to a file ..................................... 129

A Grape to Digraphs Command Map ................................. 140
  A.1 Functions to construct and modify graphs ......................... 140
  A.2 Functions to inspect graphs, vertices and edges .................. 140
  A.3 Functions to determine regularity properties of graphs ........ 141
  A.4 Some special vertex subsets of a graph ............................. 141
  A.5 Functions to construct new graphs from old ...................... 142
  A.6 Vertex-Colouring and Complete Subgraphs ....................... 142
  A.7 Automorphism groups and isomorphism testing for graphs .... 142

References ................................................................. 143

Index ................................................................. 144
Chapter 1

The Digraphs package

1.1 Introduction

This is the manual for the Digraphs package version 0.15.2. This package was developed at the University of St Andrews by:

- Jan De Beule,
- Julius Jonušas,
- James D. Mitchell,
- Michael C. Torpey, and
- Wilf A. Wilson.

Additional contributions were made by:

- Stuart Burrell,
- Luke Elliott,
- Christopher Jefferson,
- Markus Pfeiffer,
- Chris Russell, and
- Finn Smith.

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. If the **NautyTracesInterface** package for GAP is available then it is also possible to use nauty [MP14] for computing automorphism groups and canonical forms in **Digraphs**. See Section 7.2 for more details.

### 1.1.1 Definitions

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 walk** 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 $v_{i+1}$. A **directed path** is a directed walk where no vertex (and hence no edge) is repeated. A **directed circuit** is a directed walk where $v_1 = v_n$, and a **directed cycle** is a directed circuit where where no vertex is repeated, except for $v_1 = v_n$.

The length of a directed walk $(v_1, e_1, v_2, e_2, ..., e_{n-1}, v_n)$ is equal to $n - 1$, the number of edges it contains. A directed walk (or path) $(v_1, e_1, v_2, e_2, ..., e_{n-1}, v_n)$ is sometimes called a directed walk (or path) **from vertex** $v_1$ **to vertex** $v_n$. A directed walk of zero length, i.e. a sequence $(v)$ for some vertex $v$, is called **trivial**. A trivial directed walk is considered to be both a circuit and a cycle, as is the empty directed walk $(\ )$. A **simple circuit** is another name for a non-trivial and non-empty directed cycle.
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 4.9.0 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 4.5.1 or higher is available. IO must be compiled before Digraphs can be loaded.

• ensure that the Orb package version 4.8.2 or higher is available. Orb has better performance when compiled, but although compilation is recommended, it is not required to be compiled for Digraphs to 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 4.8.1 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-0.15.2.tar.gz from the Digraph package webpage.

• unzip and untar the file, this should create a directory called digraphs-0.15.2.

• locate the pkg directory of your GAP directory, which contains the directories lib, doc and so on. Move the directory digraphs-0.15.2 into the pkg directory.

• it is necessary to compile the Digraphs package. Inside the pkg/digraphs-0.15.2 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-0.15.2 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

\[\text{DigraphsMakeDoc}()\] (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

\[\text{DigraphsTestInstall}()\] (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

\[\text{DigraphsTestStandard}()\] (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

Every digraph in Digraphs belongs to the category IsDigraph. Basic attributes and operations for digraphs are: DigraphVertices (5.1.1), DigraphRange (5.2.5), DigraphSource (5.2.5), OutNeighbours (5.2.6), and DigraphEdges (5.1.3).

3.1.2 IsCayleyDigraph

IsCayleyDigraph is a subcategory of IsDigraph. Digraphs that are Cayley digraphs of a group and that are constructed by the operation CayleyDigraph (3.1.10) are constructed in this category.

3.1.3 IsDigraphWithAdjacencyFunction

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

3.1.4 DigraphType

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 (5.2.5) and DigraphRange (5.2.5).

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 $E^0 = [1 .. Length(source)]$, 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 (5.2.5) and DigraphRange (5.2.5).

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 (7.2.9), 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 \( \text{obj} \) is a binary relation on the points \([1 \ldots n]\) for some positive integer \(n\), then this function returns the digraph defined by \( \text{obj} \). 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 \( \text{obj} \).

Example

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

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

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

3.1.6 DigraphByAdjacencyMatrix

\[ \text{DigraphByAdjacencyMatrix(adj)} \]\n
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 \( \text{Length}(adj) \) vertices which has the edge \([i,j]\) if and only if \( adj[i][j] \) is true.

Example

\[
\text{gap> DigraphByAdjacencyMatrix([[}
> [0, 1, 0, 2, 0],
> [1, 1, 1, 0, 1],
> [0, 3, 2, 1, 1],
> [0, 0, 1, 0, 1],
> [2, 0, 0, 0, 0]]);
<\text{multidigraph with 5 vertices, 18 edges}>
\]

\[ \text{DigraphByEdges(edges[, n])} \]\n
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 \( n \geq m \) (with \( m \) defined as above), then this function returns the digraph with \( n \) vertices and edges \( edges \).

See DigraphEdges (5.1.3).

Example

\[
\text{gap> DigraphByEdges([[}
> [1, 3], [2, 1], [2, 3], [2, 5], [3, 6],
> [4, 6], [5, 2], [5, 4], [5, 6], [6, 6]]);
<\text{digraph with 6 vertices, 10 edges}>
\]

\[ \text{DigraphByEdges([}
> [1, 3], [2, 1], [2, 3], [2, 5], [3, 6],
> [4, 6], [5, 2], [5, 4], [5, 6], [6, 6]], 12);
<\text{digraph with 12 vertices, 10 edges}>
\]
3.1.8 EdgeOrbitsDigraph

\[ \text{EdgeOrbitsDigraph}(G, \text{edges}, n) \]

**Returns:** A new digraph.

If \( G \) is a permutation group, \( \text{edges} \) is an edge or list of edges, and \( n \) is a non-negative integer such that \( G \) fixes \([1 \ldots n]\) setwise, then this operation returns a new digraph with \( n \) vertices and the union of the orbits of the edges in \( \text{edges} \) 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);
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

\[ \text{DigraphByInNeighbours}(\text{in}) \]

**Returns:** A digraph.

If \( \text{in} \) is a list of lists of positive integers in the range \([1 \ldots \text{Length(in)}]\), then this function returns the digraph with vertices \( E^0 = [1 \ldots \text{Length(in)}] \), and edges corresponding to the entries of \( \text{in} \). More precisely, there is an edge with source vertex \( i \) and range vertex \( j \) if \( i \) is in \( \text{in}[j] \). If \( i \) occurs in \( \text{in}[j] \) with multiplicity \( k \), then there are \( k \) multiple edges from \( i \) to \( j \).

See \text{InNeighbours (5.2.7)}.

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

3.1.10 CayleyDigraph

\[ \text{CayleyDigraph}(G[, \text{gens}]) \]

**Returns:** A digraph.

Let \( G \) be any group and let \( \text{gens} \) be a list of elements of \( G \). This function returns the Cayley graph of the group with respect \( \text{gens} \). 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 \( \text{gens} \) such that \( x \ast g = y \).

If the optional second argument \( \text{gens} \) is not present, then the generators of \( G \) are used by default. The digraph created by this operation belongs to the category \text{IsCayleyDigraph (3.1.2)}, the group \( G \) can be recovered from the digraph using \text{GroupOfCayleyDigraph (5.4.1)}, and the generators \( \text{gens} \) can be obtained using \text{GeneratorsOfCayleyDigraph (5.4.2)}.

**Example**

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

Example

```gap
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, 1, 2, 3, 4 ] ])
gap> CayleyDigraph(G);
<digraph with 8 vertices, 16 edges>
gap> digraph := CayleyDigraph(G, [()]);
<digraph with 8 vertices, 8 edges>
gap> GroupOfCayleyDigraph(digraph) = G;
true
gap> GeneratorsOfCayleyDigraph(digraph);
[ () ]
```

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
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, or fail.

If `trans` is a transformation, and `n` is a non-negative integer such that the restriction of `trans` to `[1 .. n]` defines a transformation of `[1 .. n]`, then `AsDigraph` returns the functional digraph with `n` vertices defined by `trans`. See `IsFunctionalDigraph (6.1.7)`.

Specifically, the digraph returned by `AsDigraph` has `n` edges: for each vertex `x` in `[1 .. n]`, 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. If the restriction of `trans` to `[1 .. n]` does not define a transformation of `[1 .. n]`, then `AsDigraph(trans, n)` returns fail.

```
Example

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

3.2.3 Graph

\(\text{Graph}(\text{digraph})\) (operation)

Returns: A Grape package graph.

If \(\text{digraph}\) is a digraph without multiple edges, then this operation returns a Grape package graph that is isomorphic to \(\text{digraph}\).

If \(\text{digraph}\) 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 \(\text{Graph(DigraphRemoveAllMultipleEdges(digraph))}\).

Example

```gap
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, 1, 1, 1, 1, 2, 2 ])
```

```gap
gap> Digraph(Petersen);
<digraph with 10 vertices, 30 edges>
```

```gap
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, 1, 1, 1, 1, 2, 2 ])
```

3.2.4 AsGraph

\(\text{AsGraph}(\text{digraph})\) (attribute)

Returns: A Grape package graph.

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

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

Example

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

```gap
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

\[ \text{AsTransformation(digraph)} \]

\textbf{Returns:} A transformation, or fail

If \textit{digraph} is a functional digraph, then \textit{AsTransformation} returns the transformation which is defined by \textit{digraph}. See \textit{IsFunctionalDigraph} (6.1.7). Otherwise, \textit{AsTransformation(digraph)} returns fail.

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

\begin{verbatim}
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
\end{verbatim}

3.3 New digraphs from old

3.3.1 DigraphCopy

\[ \text{DigraphCopy(digraph)} \]

\textbf{Returns:} A digraph.

This function returns a new copy of \textit{digraph}, retaining none of the attributes or properties of \textit{digraph}.

\begin{verbatim}
Example

gap> gr := CycleDigraph(10);
<digraph with 10 vertices, 10 edges>
gap> DigraphCopy(gr) = gr;
true
\end{verbatim}

3.3.2 InducedSubdigraph

\[ \text{InducedSubdigraph(digraph, verts)} \]

\textbf{Returns:} A digraph.

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

The vertices of the induced subdigraph are \([1..\text{Length(verts)}]\) but the original vertex labels can be accessed via \textit{DigraphVertexLabels (5.1.9)}.
Digraphs

Example

```gap
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>
```

3.3.3 ReducedDigraph

▷ ReducedDigraph(digraph)  

**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 and edge labels of the graph are preserved. A vertex in the new digraph can be matched to the corresponding vertex in `digraph` by using the label.

The ordering of the vertices is preserved.

```gap
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, 2 ], [ ], [ 1, 3 ] ]
gap> DigraphEdges(d);
[ [ 1, 1 ], [ 1, 2 ], [ 4, 1 ], [ 4, 4 ] ]
gap> DigraphEdges(r);
[ [ 1, 1 ], [ 1, 2 ], [ 3, 1 ], [ 3, 3 ] ]
gap> DigraphVertexLabel(r, 3);
4
gap> DigraphVertexLabel(r, 2);
2
```

3.3.4 MaximalSymmetricSubdigraph

▷ MaximalSymmetricSubdigraph(digraph)  

▷ MaximalSymmetricSubdigraphWithoutLoops(digraph)  

**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 (6.1.10), IsMultiDigraph (6.1.8), and DigraphHasLoops (6.1.1) for more information.

```gap
gap> gr := Digraph([[2, 2], [1, 3], [4], [3, 1]]);
<multidigraph with 4 vertices, 7 edges>
```
Digraphs

3.3.5 MaximalAntiSymmetricSubdigraph

▷ MaximalAntiSymmetricSubdigraph(digraph)  

Returns: A digraph.

If digraph is a digraph, then MaximalAntiSymmetricSubdigraph returns a anti-symmetric sub-digraph of digraph which does not have multiple edges, has the same vertex set as digraph, and whose edge list is formed from digraph by ignoring the multiplicity of edges, and by having either an edge from the vertex u to the vertex v, or the edge from v to u (but not both) whenever both edges belong to digraph.

See IsAntiSymmetricDigraph (6.1.2) for more information.

Example

```
gap> D := Digraph([[2, 2], [1, 3], [4], [3, 1]]);
<multidigraph with 4 vertices, 7 edges>
gap> not IsAntiSymmetricDigraph(D) and IsMultiDigraph(D);
true
```

3.3.6 UndirectedSpanningTree

▷ UndirectedSpanningTree(digraph)  

▷ UndirectedSpanningForest(digraph)  

Returns: A digraph, or fail.

If digraph is a digraph with at least one vertex, then UndirectedSpanningForest returns an undirected spanning forest of digraph, otherwise this attribute returns fail. See IsUndirectedSpanningForest (4.1.2) for the definition of an undirected spanning forest.

If digraph is a digraph with at least one vertex and whose MaximalSymmetricSubdigraph (3.3.4) is connected (see IsConnectedDigraph (6.3.3)), then UndirectedSpanningTree returns an undirected spanning tree of digraph, otherwise this attribute returns fail. See IsUndirectedSpanningTree (4.1.2) for the definition of an undirected spanning tree.

Note that for a digraph that has an undirected spanning tree, the attribute UndirectedSpanningTree returns the same digraph as the attribute UndirectedSpanningForest.
Example

```gap
gap> gr := Digraph([[1, 2, 1, 3], [1], [4], [3, 4, 3]]);
<multidigraph with 4 vertices, 9 edges>
gap> UndirectedSpanningTree(gr);
fail
gap> forest := UndirectedSpanningForest(gr);
<digraph with 4 vertices, 4 edges>
gap> OutNeighbours(forest);
[[ 2 ], [ 1 ], [ 4 ], [ 3 ]]
gap> IsUndirectedSpanningForest(gr, forest);
true
gap> DigraphConnectedComponents(forest).comps;
[ [ 1, 2 ], [ 3, 4 ] ]
gap> DigraphConnectedComponents(MaximalSymmetricSubdigraph(gr)).comps;
[ [ 1, 2 ], [ 3, 4 ] ]
gap> UndirectedSpanningForest(MaximalSymmetricSubdigraph(gr))
> = forest;
true
gap> gr := CompleteDigraph(4);
<digraph with 4 vertices, 12 edges>
gap> tree := UndirectedSpanningTree(gr);
<digraph with 4 vertices, 6 edges>
gap> IsUndirectedSpanningTree(gr, tree);
true
gap> tree = UndirectedSpanningForest(gr);
true
gap> UndirectedSpanningForest(EmptyDigraph(0));
fail
```

### 3.3.7 QuotientDigraph

\[ \triangleleft \text{QuotientDigraph}(\text{digraph}, p) \]  
(\text{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
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 ] ]
```
Digraphs

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>

3.3.8 DigraphReverse

▷ DigraphReverse(digraph)  
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([[2, 4], [1], [4], [3, 4]]);
<digraph with 4 vertices, 6 edges>
gap> DigraphEdges(gr);
[ [ 1, 2 ], [ 1, 4 ], [ 2, 1 ], [ 3, 4 ], [ 4, 3 ], [ 4, 4 ] ]
gap> DigraphEdges(DigraphReverse(gr));
[ [ 1, 2 ], [ 2, 1 ], [ 3, 4 ], [ 4, 1 ], [ 4, 3 ], [ 4, 4 ] ]

3.3.9 DigraphDual

▷ DigraphDual(digraph)  
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>

3.3.10 DigraphSymmetricClosure

▷ DigraphSymmetricClosure(digraph)  
Returns: A digraph.
If \textit{digraph} is a digraph, then this attribute gives the minimal symmetric digraph which has the same vertices and contains all the edges of \textit{digraph}.

A digraph is \textit{symmetric} if its adjacency matrix \texttt{AdjacencyMatrix} (5.2.1) is symmetric. For a digraph with multiple edges this means that there are the same number of edges from a vertex \texttt{u} to a vertex \texttt{v} as there are from \texttt{v} to \texttt{u}; see \texttt{IsSymmetricDigraph} (6.1.10).

\begin{example}
\verbatim
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> List(OutNeighbours(gr1), AsSet);
[ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 4 ], [ 2, 3, 4 ] ]
\end{example}

### 3.3.11 DigraphReflexiveTransitiveClosure

\texttt{DigraphReflexiveTransitiveClosure(digraph)} (attribute)

\texttt{DigraphTransitiveClosure(digraph)} (attribute)

\textbf{Returns: } A digraph.

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

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

Let \(n\) be the number of vertices of \textit{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 \texttt{DigraphTopologicalSort} (5.1.7)], these attributes will use methods with complexity \(O(m + n + m \cdot n)\) when this is faster.

\begin{example}
\verbatim
gap> gr := DigraphFromDiSparse6String(".H\text{e}OWR\text{U}1\text{~}");
<digraph with 9 vertices, 8 edges>
gap> IsReflexiveDigraph(gr) or IsTransitiveDigraph(gr);
false
gap> OutNeighbours(gr);
[ [ 4, 6 ], [ 1, 3 ], [ ], [ 5 ], [ ], [ 7, 8, 9 ], [ ], [ ] ]
\end{example}
3.3.12 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 (3.3.11)]. 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
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
```

3.3.13 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
gap> gr := CompleteDigraph(3);
<digraph with 3 vertices, 6 edges>
```
Digraphs

```gap
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) ]) ]
```

### 3.3.14 DigraphAddVertices

- **DigraphAddVertices(digraph, m[, labels])**
  - **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.

```gap
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
```

### 3.3.15 DigraphAddEdge

- **DigraphAddEdge(digraph, edge)**
  - **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]`.

```gap
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 ] ]
```
3.3.16 DigraphAddEdgeOrbit

\[ \text{DigraphAddEdgeOrbit}(\text{digraph}, \text{edge}) \]

\textbf{Returns:} A new digraph.

This operation returns a new digraph with the same vertices and edges as \textit{digraph} and with additional edges consisting of the orbit of the edge \textit{edge} under the action of the DigraphGroup (7.2.9) of \textit{digraph}. If \textit{edge} is already an edge in \textit{digraph}, then \textit{digraph} is returned unchanged.

An edge is simply a pair of vertices of \textit{digraph}.

\begin{verbatim}
Example

\begin{small}
\text{gap> gr1 := CayleyDigraph(DihedralGroup(8));}
\text{<digraph with 8 vertices, 24 edges>}
\text{gap> gr2 := DigraphAddEdgeOrbit(gr1, [1, 8]);}
\text{<digraph with 8 vertices, 32 edges>}
\text{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 ] ]
\text{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 ] ]
\text{gap> gr3 := DigraphRemoveEdgeOrbit(gr2, [1, 8]);}
\text{<digraph with 8 vertices, 24 edges>}
\text{gap> gr3 = gr1;}
\text{true}
\end{small}
\end{verbatim}

3.3.17 DigraphAddEdges

\[ \text{DigraphAddEdges}(\text{digraph}, \text{edges}) \]

\textbf{Returns:} A digraph.

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

If an edge is included in \textit{edges} with multiplicity \textit{k}, then it will be added \textit{k} times.

\begin{verbatim}
Example

\begin{small}
\text{gap> func := function(n)
  \text{  local source, range, i;}
  \text{  source := [];
  \text{  range := [];
  \text{  for i in [1 .. n - 2] do
  \text{    Add(source, i);
  \text{    Add(range, i + 1);
  \text{  end;}
  \text{  return Digraph(n, source, range);}
  \text{end;}
\text{gap> gr := func(1024);}
\text{<digraph with 1024 vertices, 1022 edges>}
\end{small}
\end{verbatim}
Digraphs

3.3.18 DigraphRemoveVertex

DigraphRemoveVertex(digraph, v)

- **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 (5.1.9).

Example

```gap
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" ]
```

3.3.19 DigraphRemoveVertices

DigraphRemoveVertices (digraph, verts)

- **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 (5.1.9).

Example

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

3.3.20 DigraphRemoveEdge

DigraphRemoveEdge(digraph, edge)

- **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>
```

### 3.3.21 DigraphRemoveEdgeOrbit

\[\text{DigraphRemoveEdgeOrbit}(\text{digraph}, \text{edge})\]

(definition)

**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 (7.2.9) 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
```

### 3.3.22 DigraphRemoveEdges

\[\text{DigraphRemoveEdges}(\text{digraph}, \text{edges})\]

(definition)

**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 (3.3.20)].

**Example**

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

### 3.3.23 DigraphRemoveLoops

**Operation**

\[ \text{DigraphRemoveLoops}(\text{digraph}) \]

**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>
  DigraphRemoveLoops(gr);
  <digraph with 5 vertices, 8 edges>
```

### 3.3.24 DigraphRemoveAllMultipleEdges

**Operation**

\[ \text{DigraphRemoveAllMultipleEdges}(\text{digraph}) \]

**Returns:** A digraph.

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

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

### 3.3.25 DigraphReverseEdges

**Operation**

\[ \text{DigraphReverseEdges}(\text{digraph}, \text{edges}) \]

**Returns:** A digraph.

If *digraph* is a digraph without multiple edges, and *edges* is either:
• a list of pairs of vertices of \textit{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 \texttt{DigraphEdges (5.1.3)},

then \texttt{DigraphReverseEdges} returns a new digraph constructed from \textit{digraph} by reversing the orientation of every edge specified by \textit{edges}. If only one edge is to be reversed, then \texttt{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 \textit{digraph} cannot have multiple edges, the output may have multiple edges.

\begin{verbatim}
gap> gr := DigraphFromDiSparse6String(".Tg?i@s?t_e?_qEsC");
<digraph with 21 vertices, 8 edges>
gap> DigraphEdges(gr);
[ [ 1, 2 ], [ 1, 7 ], [ 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
\end{verbatim}

\section*{3.3.26 \texttt{DigraphDisjointUnion} (for an arbitrary number of digraphs)}
\begin{verbatim}
\triangleright \texttt{DigraphDisjointUnion(gr1, gr2, ...)} \hspace{1cm} \texttt{(function)}
\triangleright \texttt{DigraphDisjointUnion(list)} \hspace{1cm} \texttt{(function)}
\end{verbatim}

Returns: A digraph.

In the first form, if \texttt{gr1}, \texttt{gr2}, etc. are digraphs, then \texttt{DigraphDisjointUnion} returns their disjoint union. In the second form, if \texttt{list} is a non-empty list of digraphs, then \texttt{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 \texttt{gr1}, \texttt{gr2}, ..., the disjoint union with have \texttt{DigraphNrVertices(gr1) + DigraphNrVertices(gr2) + ...} vertices. The edges of \texttt{gr1} will remain unchanged, whilst the edges of the \textit{i}th digraph, \texttt{gr[i]}, will be changed so that they belong to the vertices of the disjoint union corresponding to \texttt{gr[i]}. In particular, the edges of \texttt{gr[i]} will have their source and range increased by \texttt{DigraphNrVertices(gr1) + ... + DigraphNrVertices(gr[i-1])}.

Note that previously set \texttt{DigraphVertexLabels (5.1.9)} will be lost.

\begin{verbatim}
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);
\end{verbatim}
Digraphs

3.3.27 DigraphEdgeUnion (for an arbitrary number of digraphs)

\[ \text{DigraphEdgeUnion(gr1, gr2, ...)} \]

\[ \text{DigraphEdgeUnion(list)} \]

\textbf{Returns:} A digraph.

In the first form, if \textit{gr1}, \textit{gr2}, etc. are digraphs, then \textit{DigraphEdgeUnion} returns their edge union.

In the second form, if \textit{list} is a non-empty list of digraphs, then \textit{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 \textit{union} of the vertex sets, whilst the edge list of the edge union is the \textit{concatenation} of the edge lists. The number of vertices of the edge union is equal to the \textit{maximum} number of vertices of one of the digraphs, whilst the number of edges of the edge union will equal the \textit{sum} of the number of edges of each digraph.

Note that previously set \textit{DigraphVertexLabels (5.1.9)} will be lost.

\textbf{Example}

\begin{verbatim}
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
\end{verbatim}

3.3.28 DigraphJoin (for an arbitrary number of digraphs)

\[ \text{DigraphJoin(gr1, gr2, ...)} \]

\[ \text{DigraphJoin(list)} \]

\textbf{Returns:} A digraph.

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

The join of a collection of digraphs \textit{gr1}, \textit{gr2}, ... is formed by first taking the \textit{DigraphDisjointUnion (3.3.26)} of the collection. In the disjoint union, if \textit{i} \neq \textit{j} then there are no edges between vertices corresponding to digraphs \textit{gr[i]} and \textit{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 (5.1.9) will be lost.

Example

```gap
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>
```

### 3.3.29 LineDigraph

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

### 3.3.30 LineUndirectedDigraph

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

### 3.3.31 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, v] and [u', v'] if and only if [u, v] is an edge in digraph, together with the original edges and their duplicates.
3.3.32 BipartiteDoubleDigraph

▷ BipartiteDoubleDigraph(digraph)

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 original 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 original edges are not included in the resulting digraph.

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

3.3.33 DigraphAddAllLoops

▷ DigraphAddAllLoops(digraph)

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.

```
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 ] ]
```

3.3.34 DistanceDigraph (for digraph and int)

▷ DistanceDigraph(digraph, i)

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

```
gap> gr := Digraph([[1, 2, 3], [1, 3], [1]]);
<digraph with 3 vertices, 6 edges>
gap> OutNeighbours(gr);
[ [ 1, 2, 3 ], [ 1, 3, 2 ], [ 1, 3 ] ]
```
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
digraph := DigraphFromSparse6String(":\n?AL'BC_DEnFB'EJaGHdIJJeGKcKL_OMcDHfILaBJfHMjKM");
gap> DistanceDigraph(digraph, 1);
<digraph with 30 vertices, 90 edges>
gap> DistanceDigraph(digraph, [1, 2]);
<digraph with 30 vertices, 270 edges>
```

3.3.35 DigraphClosure (for a digraph and positive integer)

▷ DigraphClosure(digraph, \( k \)) (operation)

**Returns:** A digraph.

Given a symmetric loopless digraph with no multiple edges digraph, the \( k \)-closure of digraph is defined to be the unique smallest symmetric loopless digraph \( C \) with no multiple edges on the vertices of digraph that contains all the edges of digraph and satisfies the property that the sum of the degrees of every two non-adjacent vertices in \( C \) is less than \( k \). See IsSymmetricDigraph (6.1.10), DigraphHasLoops (6.1.1), IsMultiDigraph (6.1.8), and OutDegreeOfVertex (5.2.10).

The operation DigraphClosure returns the \( k \)-closure of digraph.

Example

```gap
gr := CompleteDigraph(6);
gap> DigraphRemoveEdges(gr, [[1, 2], [2, 1]]);
gap> closure := DigraphClosure(gr, 6);
<digraph with 6 vertices, 30 edges>
gap> IsCompleteDigraph(closure);
true
```

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

\[ \text{RandomMultiDigraph}(n[, m]) \]

\textbf{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 \([1 \ldots \binom{n}{2}]\).

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.

```
\text{gap> RandomMultiDigraph}(1000);
<multidigraph with 1000 vertices, 216659 edges>
\text{gap> RandomMultiDigraph}(1000, 950);
<multidigraph with 1000 vertices, 950 edges>
```

3.4.3 RandomTournament

\[ \text{RandomTournament}(n) \]

\textbf{Returns:} A digraph.

If \( n \) is a non-negative integer, this function returns a random tournament with \( n \) vertices. See \text{IsTournament} (6.1.11).

```
\text{gap> RandomTournament}(10);
<digraph with 10 vertices, 45 edges>
```

3.5 Standard examples

3.5.1 ChainDigraph

\[ \text{ChainDigraph}(n) \]

\textbf{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 \text{DigraphReflexiveTransitiveClosure} (3.3.11) of a chain represents a total order.

```
\text{gap> ChainDigraph}(42);
<digraph with 42 vertices, 41 edges>
```

3.5.2 CompleteDigraph

\[ \text{CompleteDigraph}(n) \]

\textbf{Returns:} A digraph.

If \( n \) is a non-negative integer, this function returns the complete digraph with \( n \) vertices. See \text{IsCompleteDigraph} (6.1.5).

```
\text{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 that 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 \([1 \ldots m]\)) and n (containing the vertices \([m + 1 \ldots m + n]\)).

Example

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

3.5.4 CompleteMultipartiteDigraph

CompleteMultipartiteDigraph(orders) (operation)

Returns: A digraph.

For a list orders of \(n\) positive integers, this function returns the digraph containing \(n\) independent sets of vertices of orders \([l[1] \ldots l[n]]\). Moreover, each vertex is adjacent to every other not contained in the same independent set.

Example

```gap
CompleteMultipartiteDigraph([5, 4, 2]);
<digraph with 11 vertices, 76 edges>
```

3.5.5 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.

Example

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

3.5.6 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.
3.5.7 JohnsonDigraph

- **JohnsonDigraph**(\(n, k\))

  **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 \([1 \ldots k]\), and two vertices are connected by an edge iff their intersection has size \(k-1\).

\[
gap> \text{gr := JohnsonDigraph(3, 1);} \\
<\text{digraph with 3 vertices, 6 edges}> \\
gap> \text{OutNeighbours(gr);} \\
[ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ] \\
gap> \text{gr := JohnsonDigraph(4, 2);} \\
<\text{digraph with 6 vertices, 24 edges}> \\
gap> \text{OutNeighbours(gr);} \\
[ [ 2, 3, 4, 5 ], [ 1, 3, 4, 6 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ], [ 1, 3, 4, 6 ], [ 2, 3, 4, 5 ] ] \\
gap> \text{JohnsonDigraph(1, 0);} \\
<\text{digraph with 1 vertex, 0 edges}>\]
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.

4.1.1 IsSubdigraph

IsSubdigraph(super, sub)
Returns: true or false.
If super and sub are digraphs, then this operation returns true if sub is a subdigraph of super,
and false if it is not.
A digraph sub is a subdigraph of a digraph super if sub and super share the same number of
vertices, and the collection of edges of super (including repeats) contains the collection of edges of
sub (including repeats).
In other words, sub is a subdigraph of super if and only if DigraphNrVertices(sub) =
DigraphNrVertices(super), and for each pair of vertices i and j, there are at least as many edges
of the form [i, j] in super as there are in sub.

Example

gap> g := Digraph([[2, 3], [1], [2, 3]]);
<digraph with 3 vertices, 5 edges>
gap> h := Digraph([[2, 3], [], [2]]);
4.1.2 IsUndirectedSpanningTree

\[ \text{IsUndirectedSpanningTree} \text{(super, sub)} \] (operation) \n\[ \text{IsUndirectedSpanningForest} \text{(super, sub)} \] (operation) \n
\textbf{Returns:} true or false. \n
The operation IsUndirectedSpanningTree returns true if the digraph sub is an undirected spanning tree of the digraph super, and the operation IsUndirectedSpanningForest returns true if the digraph sub is an undirected spanning forest of the digraph super. \n
An \textit{undirected spanning tree} of a digraph super is a subdigraph of super that is an undirected tree (see IsSubdigraph (4.1.1) and IsUndirectedTree (6.3.8)). Note that a digraph whose MaximalSymmetricSubdigraph (3.3.4) is not connected has no undirected spanning trees (see IsConnectedDigraph (6.3.3)). \n
An \textit{undirected spanning forest} of a digraph super is a subdigraph of super that is an undirected forest (see IsSubdigraph (4.1.1) and IsUndirectedForest (6.3.8)), and is not contained in any larger such subdigraph of super. Equivalently, an undirected spanning forest is a subdigraph of super whose connected components coincide with those of the MaximalSymmetricSubdigraph (3.3.4) of super (see DigraphConnectedComponents (5.3.8)). \n
Note that an undirected spanning tree is an undirected spanning forest that is connected.

\[ \text{Example} \]

\[ \text{gap> gr := CompleteDigraph(4);} \]
\[ \text{<digraph with 4 vertices, 12 edges>} \]
\[ \text{gap> tree := Digraph([[[3, [4], [1, 4], [2, 3]]]);} \]
\[ \text{<digraph with 4 vertices, 6 edges>} \]
\[ \text{gap> IsSubdigraph(gr, tree) and IsUndirectedTree(tree);} \]
\[ \text{true} \]
\[ \text{gap> IsUndirectedSpanningTree(gr, tree);} \]
\[ \text{true} \]
\[ \text{gap> forest := EmptyDigraph(4);} \]
\[ \text{<digraph with 4 vertices, 0 edges>} \]
\[ \text{gap> IsSubdigraph(gr, forest) and IsUndirectedForest(forest);} \]
\[ \text{true} \]
\[ \text{gap> IsUndirectedSpanningForest(gr, forest);} \]
false

gap> IsSubdigraph(tree, forest);
true

gap> gr := DigraphDisjointUnion(CycleDigraph(2), CycleDigraph(2));
<digraph with 4 vertices, 4 edges>
gap> IsUndirectedTree(gr);
false

gap> IsUndirectedForest(gr) and IsUndirectedSpanningForest(gr, gr);
true
Chapter 5

Attributes and operations

5.1 Vertices and edges

5.1.1 DigraphVertices

\( \text{DigraphVertices}(\text{digraph}) \)

*Returns:* A list of integers.

Returns the vertices of the digraph \( \text{digraph} \).

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

\begin{verbatim}
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, 7, 5, 3, 7]);
<digraph with 6 vertices, 5 edges>

gap> DigraphVertices(gr);
[ 1 .. 6 ]

gap> DigraphVertices(RandomDigraph(100));
[ 1 .. 100 ]
\end{verbatim}

5.1.2 DigraphNrVertices

\( \text{DigraphNrVertices}(\text{digraph}) \)

*Returns:* An integer.

Returns the number of vertices of the digraph \( \text{digraph} \).

Example

\begin{verbatim}
gap> gr := Digraph(["a", "b", "c"],
                   ["a", "b", "b"],
                   ["b", "c", "a"]);
<digraph with 3 vertices, 3 edges>

gap> DigraphNrVertices(gr);
40
\end{verbatim}
5.1.3 DigraphEdges

\textbf{DigraphEdges}(\textit{digraph})

\textbf{Returns:} A list of lists.

\texttt{DigraphEdges} returns a list of edges of the digraph \textit{digraph}, where each edge is a pair of elements of \texttt{DigraphVertices} (5.1.1) of the form \texttt{[source,range]}.

The entries of \texttt{DigraphEdges(digraph)} are in one-to-one correspondence with the edges of \textit{digraph}. Hence \texttt{DigraphEdges(digraph)} is duplicate-free if and only if \textit{digraph} contains no multiple edges.

The entries of \texttt{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.

\begin{verbatim}
  gap> gr := DigraphFromDiSparse6String(".DaXbOe?EAM@G~");
  <multidigraph with 5 vertices, 16 edges>
  gap> edges := ShallowCopy(DigraphEdges(gr));; Sort(edges);
  gap> edges;
  [[ 1, 1 ], [ 1, 3 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 1 ],
   [ 2, 2 ], [ 2, 3 ], [ 2, 5 ], [ 3, 2 ], [ 3, 4 ], [ 3, 5 ],
   [ 4, 2 ], [ 4, 4 ], [ 4, 5 ], [ 5, 1 ]]
\end{verbatim}

5.1.4 DigraphNrEdges

\textbf{DigraphNrEdges}(\textit{digraph})

\textbf{Returns:} An integer.

This function returns the number of edges of the digraph \textit{digraph}.

\begin{verbatim}
  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
\end{verbatim}
5.1.5 DigraphSinks

\[ \texttt{DigraphSinks(digraph)} \]

**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` (5.2.10).

```
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

\[ \texttt{DigraphSources(digraph)} \]

**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` (5.2.12).

```
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

\[ \texttt{DigraphTopologicalSort(digraph)} \]

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

If `digraph` is a digraph whose only directed cycles are loops, 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 directed cycles of length greater than 1, then this operation returns `fail`.

See section 1.1.1 for the definition of a directed cycle, and the definition of a loop.

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

5.1.8 DigraphVertexLabel

\[ \texttt{DigraphVertexLabel(digraph, i)} \]

\[ \texttt{SetDigraphVertexLabel(digraph, i, obj)} \]

(operation)
If $\text{digraph}$ 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 $\text{digraph}$ to the arbitrary GAP object $\text{obj}$.

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 $\text{digraph}$ is a digraph created from a record with a component $\text{vertices}$, 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
gap> gr := DigraphFromDigraph6String("&DHUEe._");
<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"
```

5.1.9 DigraphVertexLabels

▷ $\text{DigraphVertexLabels}(\text{digraph})$  (operation)
▷ $\text{SetDigraphVertexLabels}(\text{digraph}, \text{list})$  (operation)

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

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 $\text{digraph}$ is a digraph created from a record with a component $\text{vertices}$, 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
gap> gr := DigraphFromDigraph6String("&DHUEe._");
<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);
```

Digraphs

[ "a", "d", "c" ]
gap> gr := InducedSubdigraph(gr, [1, 3]);
<digraph with 2 vertices, 0 edges>
gap> DigraphVertexLabels(gr);
[ "a", "c" ]

5.1.10 DigraphEdgeLabel

- DigraphEdgeLabel(digraph, i, j) (operation)
- SetDigraphEdgeLabel(digraph, i, j, obj)

If digraph is a digraph without multiple edges, then the first operation returns the label of the edge from vertex i to vertex j. The second operation can be used to set the label of the edge between vertex i and vertex j to the arbitrary GAP object obj.

The label of an edge can be changed an arbitrary number of times. If no label has been set for the edge, then the default value is 1.

Induced subdigraphs, and some other operations which create new digraphs from old ones, inherit their edge labels from their parents. See also DigraphEdgeLabels (5.1.11).

Example

gap> gr := DigraphFromDigraph6String("&DHUEe_");
<digraph with 5 vertices, 11 edges>
gap> DigraphEdgeLabel(gr, 3, 1);
1
gap> SetDigraphEdgeLabel(gr, 2, 5, [42]);
gap> DigraphEdgeLabel(gr, 2, 5);
[ 42 ]
gap> gr := InducedSubdigraph(gr, [2, 5]);
<digraph with 2 vertices, 3 edges>
gap> DigraphEdgeLabel(gr, 1, 2);
[ 42 ]

5.1.11 DigraphEdgeLabels

- DigraphEdgeLabels(digraph) (operation)
- SetDigraphEdgeLabels(digraph, labels)
- SetDigraphEdgeLabels(digraph, func)

If digraph is a digraph without multiple edges, then DigraphEdgeLabels returns a copy of the labels of the edges in digraph as a list of lists labels such that labels[i][j] is the label on the edge from vertex i to vertex OutNeighbours(digraph)[i][j]. SetDigraphEdgeLabels can be used to set the labels of the edges in digraph without multiple edges to the list labels of lists of arbitrary GAP objects such that list[i][j] is the label on the edge from vertex i to the vertex OutNeighbours(digraph)[i][j]. Alternatively SetDigraphEdgeLabels can be called with binary function func that as its second argument that when passed two vertices i and j returns the label for the edge between vertex i and vertex j.

The label of an edge can be changed an arbitrary number of times. If no label has been set for an edge, then the default value is 1.
Induced subdigraphs, and some other operations which create new digraphs from old ones, inherit their labels from their parents.

Example

```gap
> gr := DigraphFromDigraph6String("&DHUEe_");
<digraph with 5 vertices, 11 edges>
> DigraphEdgeLabels(gr);
[ [ 1 ], [ 1, 1, 1 ], [ 1 ], [ 1, 1, 1 ], [ 1, 1, 1 ] ]
> SetDigraphEdgeLabel(gr, 2, 1, "d");
> DigraphEdgeLabels(gr);
[ [ 1 ], [ "d", 1, 1 ], [ 1 ], [ 1, 1, 1 ], [ 1, 1, 1 ] ]
> gr := InducedSubdigraph(gr, [1, 2, 3]);
<digraph with 3 vertices, 4 edges>
> DigraphEdgeLabels(gr);
[ [ 1 ], [ "d", 1 ], [ 1 ] ]
> OutNeighbours(gr);
[ [ 3 ], [ 1, 3 ], [ 1 ] ]
```

5.1.12 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>
> DigraphInEdges(gr, 2);
[ [ 1, 2 ], [ 1, 2 ] ]
```

5.1.13 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>
> DigraphOutEdges(gr, 2);
[ [ 2, 3 ], [ 2, 3 ] ]
```

5.1.14 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.
Digraphs

Example

```gap
gap> gr := Digraph([[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
```

5.1.15 IsMatching

▷ IsMatching(digraph, list) (operation)
▷ IsMaximalMatching(digraph, list) (operation)
▷ IsPerfectMatching(digraph, list) (operation)

Returns: true or false.

If `digraph` is a digraph and `list` is a list of pairs of vertices of `digraph`, then `IsMatching` returns true if `list` is a matching of `digraph`. The operations `IsMaximalMatching` and `IsPerfectMatching` return true if `list` is a maximal, or perfect, matching of `digraph`, respectively. Otherwise, these operations return false.

A matching `M` of a digraph `digraph` is a subset of the edges of `digraph`, i.e. `DigraphEdges(digraph)`, such that no pair of distinct edges in `M` are incident to the same vertex of `digraph`. Note that this definition allows a matching to contain loops. See `DigraphHasLoops(6.1.1)`. The matching `M` is maximal if it is contained in no larger matching of the digraph, and is perfect if every vertex of the digraph is incident to an edge in the matching. Every perfect matching is maximal.

Example

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

```

```gap
gap> IsMatching(gr, edges);
true
gap> IsMaximalMatching(gr, edges);
true
```

```gap
gap> edges := [[5, 1], [3, 3]];

```

```gap
gap> IsPerfectMatching(gr, edges);
true
```

```gap
gap> edges := [[1, 2], [3, 3], [4, 5]];

```

```gap
gap> IsPerfectMatching(gr, edges);
true
```
5.2 Neighbours and degree

5.2.1 AdjacencyMatrix

\[ \text{AdjacencyMatrix}(\text{digraph}) \]  
\[ \text{AdjacencyMatrixMutableCopy}(\text{digraph}) \]

**Returns:** A square matrix of non-negative integers.

This function returns the adjacency matrix \( \text{mat} \) of the digraph \( \text{digraph} \). The value of the matrix entry \( \text{mat}[i][j] \) is the number of edges in \( \text{digraph} \) with source \( i \) and range \( j \). If \( \text{digraph} \) has no vertices, then the empty list is returned.

The function \( \text{AdjacencyMatrix} \) returns an immutable list of immutable lists, whereas the function \( \text{AdjacencyMatrixMutableCopy} \) returns a copy of \( \text{AdjacencyMatrix} \) that is a mutable list of mutable lists.

**Example**

```gap
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, 1, 0, 1, 1, 1, 1], > [0, 0, 0, 1, 0, 1, 1, 0, 0, 0], > [1, 1, 1, 0, 0, 0, 1, 0, 1, 0], > [0, 0, 2, 0, 0, 0, 0, 0, 0, 0], > [1, 0, 1, 0, 1, 0, 0, 1, 0, 1], > [1, 0, 0, 0, 0, 0, 0, 0, 0, 0], > [0, 1, 1, 1, 0, 1, 0, 1, 0, 1], > [0, 1, 1, 1, 0, 1, 0, 0, 1, 1], > [0, 1, 1, 1, 0, 1, 0, 0, 1, 1]]
```

5.2.2 CharacteristicPolynomial

\[ \text{CharacteristicPolynomial}(\text{digraph}) \]

**Returns:** A polynomial with integer coefficients.

This function returns the characteristic polynomial of the digraph \( \text{digraph} \). That is it returns the characteristic polynomial of the adjacency matrix of the digraph \( \text{digraph} \).

**Example**

```gap
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> CharacteristicPolynomial(gr); \[ x_1^{10}-3*x_1^9+7*x_1^8+x_1^7+14*x_1^6+51*x_1^5-26*x_1^4+51*x_1^3-10*x_1^2+18*x_1-30 \]
```

```gap
gap> gr := CompleteDigraph(5); <digraph with 5 vertices, 20 edges>
```
Digraphs

CharacteristicPolynomial(gr);

\[ x_1^5 - 10x_1^3 - 20x_1^2 - 15x_1 - 4 \]

### 5.2.3 BooleanAdjacencyMatrix

**BooleanAdjacencyMatrix**

- **BooleanAdjacencyMatrix(digraph)** (attribute)
- **BooleanAdjacencyMatrixMutableCopy(digraph)** (operation)

**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 \( i \). If *digraph* has no vertices, then the empty list is returned.

Note that the boolean adjacency matrix loses information about multiple edges.

The attribute `BooleanAdjacencyMatrix` returns an immutable list of immutable lists, whereas the function `BooleanAdjacencyMatrixMutableCopy` returns a copy of the `BooleanAdjacencyMatrix` that is a mutable list of mutable lists.

**Example**

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

### 5.2.4 DigraphAdjacencyFunction

**DigraphAdjacencyFunction**

- **DigraphAdjacencyFunction(digraph)** (attribute)

**Returns:** A function.

If *digraph* is a digraph, then `DigraphAdjacencyFunction(digraph)` 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
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
```
Digraphs

```gap
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
```

```
5.2.5 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`.

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

Example: Example

```gap
gap> DigraphRange(gr);
[ 2, 1, 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, 2], [1, 1], [1, 3], [1, 5], [2, 1], [2, 3],
  [2, 4], [3, 2], [3, 3], [4, 2], [5, 1], [5, 2],
  [5, 3], [5, 4] ]
```

5.2.6 OutNeighbours

• OutNeighbours(digraph) (attribute)
• OutNeighbors(digraph) (attribute)
• OutNeighboursMutableCopy(digraph) (operation)
• OutNeighborsMutableCopy(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 OutNeighboursMutableCopy returns a copy of OutNeighbours which is a mutable list of mutable lists.
Digraphs

Example

```gap
> gr := Digraph(["a", "b", "c"],
>                  ["a", "b", "b"],
>                  ["b", "a", "c"]);
<digraph with 3 vertices, 3 edges>
> OutNeighbours(gr);
[ [ 2 ], [ 1, 3 ], [ ] ]
> gr := Digraph([[1, 2, 3], [2, 1], [3]]);
<digraph with 3 vertices, 6 edges>
> OutNeighbours(gr);
[ [ 1, 2, 3 ], [ 2, 1 ], [ 3 ] ]
> gr := DigraphByAdjacencyMatrix([[
> [1, 2, 1],
> [1, 1, 0],
> [0, 0, 1]]);
<multidigraph with 3 vertices, 7 edges>
> OutNeighboursMutableCopy(gr);
[ [ 1, 2, 2, 3 ], [ 1, 2 ], [ 3 ] ]
```

5.2.7 InNeighbours

▷ InNeighbours(digraph) (attribute)
▷ InNeighbors(digraph) (attribute)
▷ InNeighboursMutableCopy(digraph) (operation)
▷ InNeighborsMutableCopy(digraph) (operation)

**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.

The function InNeighbours returns an immutable list of immutable lists, whereas the function InNeighboursMutableCopy returns a copy of InNeighbours which is a mutable list of mutable lists.

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>
> InNeighbours(gr);
[ [ 2 ], [ 1 ], [ 2 ] ]
> gr := Digraph([[1, 2, 3], [2, 1], [3]]);
<digraph with 3 vertices, 6 edges>
> InNeighbours(gr);
[ [ 1, 2, 3 ], [ 1, 2 ], [ 3 ] ]
> gr := DigraphByAdjacencyMatrix([[
> [1, 2, 1],
> [1, 1, 0],
> [0, 0, 1]]);
<multidigraph with 3 vertices, 7 edges>
> InNeighboursMutableCopy(gr);
[ [ 1, 2, 2, 3 ], [ 1, 2 ], [ 3 ] ]
```
<multidigraph with 3 vertices, 7 edges>
\texttt{gap> InNeighbours(gr);}
\begin{verbatim}
[ [ 1, 2 ], [ 1, 1, 2 ], [ 1, 3 ] ]
\end{verbatim}
\texttt{gap> InNeighboursMutableCopy(gr);}
\begin{verbatim}
[ [ 1, 2 ], [ 1, 1, 2 ], [ 1, 3 ] ]
\end{verbatim}

5.2.8 OutDegrees

\begin{itemize}
\item \texttt{OutDegrees(digraph)} (attribute)
\item \texttt{OutDegreeSequence(digraph)} (attribute)
\item \texttt{OutDegreeSet(digraph)} (attribute)
\end{itemize}

\textbf{Returns:} A list of non-negative integers.

Given a digraph \texttt{digraph} with \( n \) vertices, the function \texttt{OutDegrees} returns a list \texttt{out} of length \( n \), such that for a vertex \( i \) in \texttt{digraph}, the value of \texttt{out}[i] is the out-degree of vertex \( i \). See \texttt{OutDegreeOfVertex (5.2.10)}.

The function \texttt{OutDegreeSequence} returns the same list, after it has been sorted into non-increasing order.

The function \texttt{OutDegreeSet} returns the same list, sorted into increasing order with duplicate entries removed.

\textbf{Example}
\begin{verbatim}
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 ]
\end{verbatim}

5.2.9 InDegrees

\begin{itemize}
\item \texttt{InDegrees(digraph)} (attribute)
\item \texttt{InDegreeSequence(digraph)} (attribute)
\item \texttt{InDegreeSet(digraph)} (attribute)
\end{itemize}

\textbf{Returns:} A list of non-negative integers.

Given a digraph \texttt{digraph} with \( n \) vertices, the function \texttt{InDegrees} returns a list \texttt{inn} of length \( n \), such that for a vertex \( i \) in \texttt{digraph}, the value of \texttt{inn}[i] is the in-degree of vertex \( i \). See \texttt{InDegreeOfVertex (5.2.12)}.

The function \texttt{InDegreeSequence} returns the same list, after it has been sorted into non-increasing order.

The function \texttt{InDegreeSet} returns the same list, sorted into increasing order with duplicate entries removed.

\textbf{Example}
\begin{verbatim}
gap> gr := Digraph([[1, 3, 2, 2, 4], [], [2, 1], []]);
<multidigraph with 4 vertices, 8 edges>
gap> InDegrees(gr);
[ 2, 3, 1, 2 ]
gap> InDegreeSequence(gr);
\end{verbatim}
5.2.10 OutDegreeOfVertex

\> OutDegreeOfVertex(digraph, vertex)  
\hspace{1cm} (operation)

**Returns:** The non-negative integer.

This operation returns the out-degree of the vertex \(\text{vertex}\) in the digraph \(\text{digraph}\). The out-degree of \(\text{vertex}\) is the number of edges in \(\text{digraph}\) whose source is \(\text{vertex}\).

```
\text{gap> gr := Digraph([}
  > [2, 2, 1], [1, 4], [2, 2, 4, 2], [1, 1, 2, 2, 1, 2, 2]);}
\langle\text{multidigraph with 4 vertices, 16 edges}\rangle
\text{gap> OutDegreeOfVertex(gr, 1);}  3
\text{gap> OutDegreeOfVertex(gr, 2);}  2
\text{gap> OutDegreeOfVertex(gr, 3);}  4
\text{gap> OutDegreeOfVertex(gr, 4);}  7
```

5.2.11 OutNeighboursOfVertex

\> OutNeighboursOfVertex(digraph, vertex)  
\> OutNeighborsOfVertex(digraph, vertex)  
\hspace{1cm} (operation)

**Returns:** A list of vertices.

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

```
\text{gap> gr := Digraph([}
  > [2, 2, 3], [1, 3, 4], [2, 2, 3], [1, 1, 2, 2, 1, 2, 2]);}
\langle\text{multidigraph with 4 vertices, 16 edges}\rangle
\text{gap> OutNeighboursOfVertex(gr, 1);}  \{2, 2, 3\}
\text{gap> OutNeighboursOfVertex(gr, 3);}  \{2, 2, 3\}
```

5.2.12 InDegreeOfVertex

\> InDegreeOfVertex(digraph, vertex)  
\hspace{1cm} (operation)

**Returns:** A non-negative integer.

This operation returns the in-degree of the vertex \(\text{vertex}\) in the digraph \(\text{digraph}\). The in-degree of \(\text{vertex}\) is the number of edges in \(\text{digraph}\) whose range is \(\text{vertex}\).

```
\text{gap> gr := Digraph([}
  > [2, 2, 1], [1, 4], [2, 2, 4, 2], [1, 1, 2, 2, 1, 2, 2]);}
```
5.2.13 InNeighboursOfVertex

- InNeighboursOfVertex(digraph, vertex) (operation)

Returns: A list of positive vertices.

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

Example

```gap
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 ]
```

5.2.14 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
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 ]
```

5.2.15 PartialOrderDigraphMeetOfVertices (for a digraph and two vertices)

- PartialOrderDigraphMeetOfVertices(digraph, u, v) (operation)
- PartialOrderDigraphJoinOfVertices(digraph, u, v) (operation)

Returns: A positive integer or fail

Example
If the first argument is a partial order digraph IsPartialOrderDigraph (6.1.14) then these operations return the meet, or the join, of the two input vertices. If the meet (or join) is does not exist then fail is returned. The meet (or join) is guaranteed to exist when the first argument satisfies IsMeetSemilatticeDigraph (6.1.15) (or IsJoinSemilatticeDigraph (6.1.15)) - see the documentation for these properties for the definition of the meet (or the join).

Example

```gap
gap> gr := Digraph([[1], [1, 2], [1, 3], [1, 2, 3, 4]]);
<digraph with 4 vertices, 9 edges>
gap> PartialOrderDigraphMeetOfVertices(gr, 2, 3); 4
4
gap> PartialOrderDigraphJoinOfVertices(gr, 2, 3); 1
1
gap> PartialOrderDigraphMeetOfVertices(gr, 1, 2); 2
2
gap> PartialOrderDigraphJoinOfVertices(gr, 3, 4); 3
3
gap> gr := Digraph([[1], [2], [1, 2, 3], [1, 2, 4]]);
<digraph with 4 vertices, 8 edges>
gap> PartialOrderDigraphMeetOfVertices(gr, 3, 4); fail
fail
gap> PartialOrderDigraphJoinOfVertices(gr, 3, 4); fail
fail
gap> PartialOrderDigraphMeetOfVertices(gr, 1, 2); fail
fail
gap> PartialOrderDigraphJoinOfVertices(gr, 1, 2); fail
fail
```

5.3 Reachability and connectivity

5.3.1 DigraphDiameter

\(\texttt{DigraphDiameter(digraph)}\)

Returns: An integer or fail.

This function returns the diameter of the digraph \textit{digraph}.

If a digraph \textit{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 \textit{digraph} is undefined, and this function returns the value \texttt{fail}.

See \texttt{DigraphShortestDistances (5.3.3)}.

Example

```gap
gap> gr := Digraph([[2], [3], [4, 5], [5], [1, 2, 3, 4, 5]]);
<digraph with 5 vertices, 10 edges>
gap> DigraphDiameter(gr); 3
3
gap> gr := ChainDigraph(2);
<digraph with 2 vertices, 1 edge>
gap> DigraphDiameter(gr); fail
fail
gap> IsStronglyConnectedDigraph(gr); false
false
```
5.3.2 DigraphShortestDistance (for a digraph and two vertices)

\[ \text{DigraphShortestDistance} \left( \text{digraph}, u, v \right) \]

\[ \text{DigraphShortestDistance} \left( \text{digraph} \right) \]

\[ \text{DigraphShortestDistance} \left( \text{digraph}, \text{list} \right) \]

\[ \text{DigraphShortestDistance} \left( \text{digraph}, \text{list1}, \text{list2} \right) \]

**Returns:** An integer or fail

If there is a directed path in the digraph \( \text{digraph} \) between vertex \( u \) and vertex \( v \), then this operation returns the length of the shortest such directed path. If no such directed path exists, then this operation returns fail. See section 1.1.1 for the definition of a directed path.

If the second form is used, then \( \text{list} \) 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 \( \text{list1} \) and \( \text{list2} \) are both lists of vertices. The shortest directed path between \( \text{list1} \) and \( \text{list2} \) is then the length of the shortest directed path which starts with a vertex in \( \text{list1} \) and terminates at a vertex in \( \text{list2} \), if such directed path exists. If \( \text{list1} \) and \( \text{list2} \) have non-empty intersection, the operation returns 0.

**Example**

```
gap> gr := Digraph([[2], [3], [1, 4], [1, 3], [5]]);
gap> DigraphShortestDistance(gr, 1, 3);
2
```

5.3.3 DigraphShortestDistances

\[ \text{DigraphShortestDistances} \left( \text{digraph} \right) \]

**Returns:** A square matrix.

If \( \text{digraph} \) is a digraph with \( n \) vertices, then this function returns an \( n \times n \) matrix \( \text{mat} \), 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 \( \text{mat}[i][j] \) is the length of the shortest such directed path. If no such directed path exists, then the value of \( \text{mat}[i][j] \) is fail. We use the convention that the distance from every vertex to itself is 0, i.e. \( \text{mat}[i][i] = 0 \) 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]]);
gap> mat := DigraphShortestDistances(gr);
gap> PrintArray(mat);
[ [ 0, 1, 2, fail ],
  [ 2, 0, 1, fail ],
  [ fail, fail, 0, fail ],
  [ fail, fail, fail, 0 ]]
```
5.3.4 DigraphLongestDistanceFromVertex

Operation

DigraphLongestDistanceFromVertex(digraph, v)

Returns: An integer, or infinity.

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. See section 1.1.1 for the definitions of directed walk, directed cycle, and loop.

- If there exists a directed walk starting at vertex v which traverses a loop or a directed cycle, then we consider there to be a walk of infinite length from v (realised by repeatedly traversing the loop/directed cycle), and so the result is infinity. To disallow walks using loops, try using DigraphRemoveLoops (3.3.23):
  DigraphLongestDistanceFromVertex(DigraphRemoveLoops(digraph, v)).

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

Note that the result is 0 if and only if v is a sink of digraph. See DigraphSinks (5.1.5).

Example

```gap
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)

Operation

DigraphDistanceSet(digraph, vertex, distance)

DigraphDistanceSet(digraph, vertex, distances)

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
gap> gr := Digraph([[2], [3, 4], [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 ]```
5.3.6 DigraphGirth

\[ \text{DigraphGirth}(\text{digraph}) \]  
\hspace{1em} (attribute)

**Returns:** An integer, or infinity.

This attribute returns the *girth* of the digraph *digraph*. The *girth* of a digraph is the length of its shortest simple circuit. See section 1.1.1 for the definitions of simple circuit, directed cycle, and loop.

If *digraph* has no directed 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 (7.2.9)], then the performance is likely to be better.

For symmetric digraphs, see also DigraphUndirectedGirth (5.3.7).

\[ \text{Example} \]

\begin{verbatim}
gap> gr := Digraph([[1], [1]]);
<digraph with 2 vertices, 2 edges>
gap> DigraphGirth(gr);
1
gap> gr := Digraph([[2, 3], [3], [4], [1]]);
<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
\end{verbatim}

5.3.7 DigraphUndirectedGirth

\[ \text{DigraphUndirectedGirth}(\text{digraph}) \]  
\hspace{1em} (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 or edge more than once.

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

\[ \text{Example} \]

\begin{verbatim}
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);
\end{verbatim}
infinity

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

### 5.3.8 DigraphConnectedComponents

**DigraphConnectedComponents(digraph)**

**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 directed path (ignoring the orientation of edges) between them. More formally, 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 (3.3.10)` 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.

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

### 5.3.9 DigraphConnectedComponent

**DigraphConnectedComponent(digraph, vertex)**

**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.

```gap
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 ]
```
5.3.10 DigraphStronglyConnectedComponents

\texttt{DigraphStronglyConnectedComponents(digraph)}

\textbf{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: \( \text{comps} \) and \( \text{id} \). The component \( \text{comps} \) is a list of the strongly connected components of \( digraph \) (each of which is a list of vertices). The component \( \text{id} \) is a list such that the vertex \( i \) is an element of the strongly connected component \( \text{comps}[\text{id}[i]] \).

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.

\texttt{gap> gr := Digraph([[2], [3, 1], [ ]]);
<digraph with 3 vertices, 3 edges>
gap> DigraphStronglyConnectedComponents(gr);
rec( comps := [ [ 3 ], [ 1, 2 ] ], id := [ 2, 2, 1 ] )}

5.3.11 DigraphStronglyConnectedComponent

\texttt{DigraphStronglyConnectedComponent(digraph, vertex)}

\textbf{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 \texttt{DigraphStronglyConnectedComponents} (5.3.10) for more information.

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

5.3.12 DigraphBicomponents

\texttt{DigraphBicomponents(digraph)}

\textbf{Returns:} A pair of lists of vertices, or fail.

If \( digraph \) is a bipartite digraph, i.e. if it satisfies \texttt{IsBipartiteDigraph} (6.1.3), then \texttt{DigraphBicomponents} returns a pair of bicomponents of \( digraph \). Otherwise, \texttt{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 (6.3.3) 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 ] ]

5.3.13 ArticulationPoints

▷ ArticulationPoints(digraph) (attribute)

Returns: A list of vertices.

A connected digraph is biconnected if it is still connected (in the sense of IsConnectedDigraph (6.3.3)) when any vertex is removed. If the digraph digraph is not biconnected but is connected, then any vertex v of digraph whose removal makes the resulting digraph disconnected is called an articulation point.

ArticulationPoints returns a list of the articulation points of digraph, if any, and, in particular, returns the empty list if digraph is not connected.

Multiple edges and loops are ignored by this method.

The method used in this operation has complexity $O(m + n)$ where $m$ is the number of edges (counting multiple edges as one, and not counting loops) and $n$ is the number of vertices in the digraph. See also IsBiconnectedDigraph (6.3.4).

Example

gap> ArticulationPoints(CycleDigraph(5));
[ ]
gap> digraph := Digraph([[2, 7], [3, 5], [4], [2], [6], [1], []]);
gap> ArticulationPoints(digraph);
[ 2, 1 ]
gap> ArticulationPoints(ChainDigraph(5));
[ 4, 3, 2 ]
gap> ArticulationPoints(NullDigraph(5));
[ ]

5.3.14 DigraphPeriod

▷ DigraphPeriod(digraph) (attribute)

Returns: An integer.
This function returns the period of the digraph \textit{digraph}.

If a digraph \textit{digraph} has at least one directed cycle, then the period is the greatest positive integer which divides the lengths of all directed cycles of \textit{digraph}. If \textit{digraph} has no directed cycles, then this function returns 0. See section 1.1.1 for the definition of a directed cycle.

A digraph with a period of 1 is said to be \textit{aperiodic}. See \textit{IsAperiodicDigraph} (6.3.6).

\begin{verbatim}
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 := ChainDigraph(2);
 <digraph with 2 vertices, 1 edge>
 gap> DigraphPeriod(gr);
 0
 gap> IsAcyclicDigraph(gr);
 true
\end{verbatim}

5.3.15 \textbf{DigraphFloydWarshall}

\textbf{DigraphFloydWarshall} is customised by the arguments \textit{func}, \textit{nopath}, and \textit{edge}. The arguments \textit{nopath} and \textit{edge} can be arbitrary \textsc{gap} objects. The argument \textit{func} must be a function which accepts 4 arguments: the matrix \textit{mat}, followed by 3 positive integers. The function \textit{func} is where the work to calculate the desired outcome must be performed.

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

\begin{verbatim}
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;
\end{verbatim}

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

\begin{verbatim}
Example
 gap> gr := DigraphFromDigraph6String("&EAHQeDB");
 <digraph with 6 vertices, 12 edges>
\end{verbatim}
Digraphs

```gap
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 ] ]
```

### 5.3.16 IsReachable

▷ **IsReachable(digraph, u, v)**

*Returns:* true or false.

This operation returns true if there exists a non-trivial directed walk from vertex u to vertex v in the digraph *digraph*, and false if there does not exist such a directed walk. See section 1.1.1 for the definition of a non-trivial directed walk.

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
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
```

### 5.3.17 DigraphPath

▷ **DigraphPath(digraph, u, v)**

*Returns:* A pair of lists, or fail.

If there exists a non-trivial directed path (or a non-trivial cycle, in the case that $u = v$) from vertex $u$ to vertex $v$ in the digraph *digraph*, then this operation returns such a directed path (or directed cycle). Otherwise, this operation returns *fail*. See Section 'Definitions' for the definition of a directed path and a directed cycle.

A directed path (or directed cycle) of non-zero length $n-1$, $(v_1, e_1, v_2, e_2, ..., e_{n-1}, v_n)$, is represented by a pair of lists $[v, a]$ as follows:

- $v$ is the list $[v_1, v_2, ..., v_n]$. 
• a is the list of positive integers \([a_1, a_2, ..., a_{n-1}]\) where for each each \(i < n\), \(a_i\) is the position of \(v_{i+1}\) in \(\text{OutNeighboursOfVertex}(\text{digraph}, v_i)\) corresponding to the edge \(e_i\). This is can be useful if the position of a vertex in a list of out-neighbours is significant, for example in orbit digraphs.

The method for \(\text{DigraphPath}\) has worst case complexity of \(O(m + n)\) where \(m\) is the number of edges and \(n\) the number of vertices in \(\text{digraph}\).

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{gap}&gt; gr := \text{Digraph([[2], [3], [2, 3]]);}</td>
</tr>
<tr>
<td>\text{&lt;digraph with 3 vertices, 4 edges&gt;}</td>
</tr>
<tr>
<td>\text{gap}&gt; \text{DigraphPath(gr, 1, 3);}</td>
</tr>
<tr>
<td>\text{[[1, 2, 3], [1, 1]]}</td>
</tr>
<tr>
<td>\text{gap}&gt; \text{DigraphPath(gr, 2, 1);}</td>
</tr>
<tr>
<td>fail</td>
</tr>
<tr>
<td>\text{gap}&gt; \text{DigraphPath(gr, 3, 3);}</td>
</tr>
<tr>
<td>\text{[[3, 3], [2]]}</td>
</tr>
<tr>
<td>\text{gap}&gt; \text{DigraphPath(gr, 1, 1);}</td>
</tr>
<tr>
<td>fail</td>
</tr>
</tbody>
</table>

5.3.18 \text{DigraphShortestPath}

\(\text{DigraphShortestPath(digraph, u, v)}\)  
\textbf{Returns:} A pair of lists, or \text{fail}.

Returns the shortest directed path in the digraph \(\text{digraph}\) from the vertex \(u\) to the vertex \(v\), if such a path exists. If \(u = v\), then the shortest non-trivial cycle is returned, again, if it exists. Otherwise, this operation returns \text{fail}. See Section ‘Definitions’ for the definition of a directed path and a directed cycle.

See \(\text{DigraphPath}\) (5.3.17) for details on the output. The method for \(\text{DigraphShortestPath}\) has worst case complexity of \(O(m + n)\) where \(m\) is the number of edges and \(n\) the number of vertices in \(\text{digraph}\).

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{gap}&gt; gr := \text{Digraph([[1, 2], [3], [2, 4], [1], [2, 4]]);}</td>
</tr>
<tr>
<td>\text{&lt;digraph with 5 vertices, 8 edges&gt;}</td>
</tr>
<tr>
<td>\text{gap}&gt; \text{DigraphShortestPath(gr, 5, 1);}</td>
</tr>
<tr>
<td>\text{[[5, 4, 1], [2, 1]]}</td>
</tr>
<tr>
<td>\text{gap}&gt; \text{DigraphShortestPath(gr, 3, 3);}</td>
</tr>
<tr>
<td>\text{[[3, 2, 3], [1, 1]]}</td>
</tr>
<tr>
<td>\text{gap}&gt; \text{DigraphShortestPath(gr, 5, 5);}</td>
</tr>
<tr>
<td>fail</td>
</tr>
<tr>
<td>\text{gap}&gt; \text{DigraphShortestPath(gr, 1, 1);}</td>
</tr>
<tr>
<td>\text{[[1, 1], [1]]}</td>
</tr>
</tbody>
</table>

5.3.19 \text{IteratorOfPaths}

\(\text{IteratorOfPaths(digraph, u, v)}\)  
\textbf{Returns:} An iterator.

If \(\text{digraph}\) is a digraph or a list of adjacencies which defines a digraph - see \(\text{OutNeighbours}\) (5.2.6) - then this operation returns an iterator of the non-trivial directed paths (or directed cycles, in the case that \(u = v\)) in \(\text{digraph}\) from the vertex \(u\) to the vertex \(v\).
See DigraphPath (5.3.17) for more information about the representation of a directed path or directed cycle which is used, and see (Reference: Iterators) for more information about iterators. See Section ‘Definitions’ for the definition of a directed path and a directed cycle.

Example

```gap
gap> gr := Digraph([[1, 4, 2], [3, 5], [2, 3], [1, 2], [4]]);
<multidigraph with 5 vertices, 11 edges>
gap> iter := IteratorOfPaths(gr, 1, 4);
<iterator>
gap> NextIterator(iter);
[ [ 1, 4 ], [ 2 ] ]
gap> NextIterator(iter);
[ [ 1, 4 ], [ 3 ] ]
gap> NextIterator(iter);
[ [ 1, 2, 5, 4 ], [ 4, 2, 1 ] ]
gap> IsDoneIterator(iter);
true
gap> iter := IteratorOfPaths(gr, 4, 3);
<iterator>
gap> NextIterator(iter);
[ [ 4, 1, 2, 3 ], [ 1, 4, 1 ] ]
```

5.3.20 DigraphAllSimpleCircuits

▷ DigraphAllSimpleCircuits(digraph)

Returns: A list of lists of vertices.

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

See section 1.1.1 for the definition of a simple circuit, and related notions. Note that a loop is a simple circuit.

For a digraph without multiple edges, a simple circuit is uniquely determined by its subsequence of vertices. However this is not the case for a multidigraph. The attribute DigraphAllSimpleCircuits ignores multiple edges, and identifies a simple circuit using only its subsequence of vertices. For example, although the simple circuits \((v, e, v)\) and \((v, e', v)\) (for distinct edges \(e\) and \(e'\)) are mathematically distinct, DigraphAllSimpleCircuits considers them to be the same.

With this approach, a directed circuit of length \(n\) can be determined by a list of its first \(n\) vertices. Thus a simple circuit \((v_1, e_1, v_2, \ldots, e_n, v_n)\) can be represented as the list \([v_1, \ldots, v_n]\), or any cyclic permutation thereof. For each simple circuit of digraph, DigraphAllSimpleCircuits(digraph) includes precisely one such list to represent the circuit.

Example

```gap
gap> gr := Digraph([[1, 4, 2], [3, 5], [2, 3], [1, 2], [4]]);
<multidigraph with 5 vertices, 11 edges>
gap> DigraphAllSimpleCircuits(gr);
[ [ 4 ], [ 4, 5 ] ]
gap> gr := ChainDigraph(10);
<digraph with 5 vertices, 6 edges>
gap> DigraphAllSimpleCircuits(gr);
[ ]
gap> gr := Digraph([[3], [1], [1]]);
<digraph with 3 vertices, 3 edges>
gap> DigraphAllSimpleCircuits(gr);
[ [ 1, 3 ] ]
```
Digraphs

```gap
gap> gr := Digraph([ [1, 1] ]);  
<multidigraph with 1 vertex, 2 edges>
gap> DigraphAllSimpleCircuits(gr);  
[ [ 1 ] ]
```

5.3.21 DigraphLongestSimpleCircuit

riter: *DigraphLongestSimpleCircuit(digraph)*

Returns: A list of vertices, or fail.

If *digraph* is a digraph, then *DigraphLongestSimpleCircuit* returns the longest simple circuit in *digraph*. See section 1.1.1 for the definition of simple circuit, and the definition of length for a simple circuit.

This attribute computes *DigraphAllSimpleCircuits(digraph)* to find all the simple circuits of *digraph*, and returns one of maximal length. A simple circuit is represented as a list of vertices, in the same way as described in *DigraphAllSimpleCircuits* (5.3.20).

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

```gap
gap> gr := CompleteDigraph(4);;  
gap> DigraphLayers(gr, 1);  
```

5.3.22 DigraphLayers

riter: *DigraphLayers(digraph, vertex)*

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.

```gap
gap> gr := CompleteDigraph(4);;  
gap> DigraphLayers(gr, 1);  
```

5.3.23 DigraphDegeneracy

riter: *DigraphDegeneracy(digraph)*

Returns: A non-negative integer, or fail.

If *digraph* is a symmetric digraph without multiple edges - see *IsSymmetricDigraph* (6.1.10) and *IsMultiDigraph* (6.1.8) - then this attribute returns the degeneracy of *digraph*.

```gap
gap> gr := ChainDigraph(10);;  
gap> DigraphLongestSimpleCircuit(gr);  
fail
```
The degeneracy of a digraph is the least integer \( k \) such that every induced of \( \text{digraph} \) contains a vertex whose number of neighbours (excluding itself) is at most \( k \). Note that this means that loops are ignored.

If \( \text{digraph} \) is not symmetric or has multiple edges then this attribute returns \text{fail}. 

```
  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
```

### 5.3.24 DigraphDegeneracyOrdering

\[ \text{DigraphDegeneracyOrdering}(\text{digraph}) \]

**Returns:** A list of integers, or \text{fail}.

If \( \text{digraph} \) is a digraph for which \( \text{DigraphDegeneracy}(\text{digraph}) \) is a non-negative integer \( k \) - see \( \text{DigraphDegeneracy} \) (5.3.23) - then this attribute returns a degeneracy ordering of the vertices of \( \text{digraph} \).

A degeneracy ordering of \( \text{digraph} \) is a list ordering of the vertices of \( \text{digraph} \) ordered such that for any position \( i \) of the list, the vertex \( \text{ordering}[i] \) has at most \( k \) neighbours in later position of the list.

If \( \text{DigraphDegeneracy}(\text{digraph}) \) returns \text{fail}, then this attribute returns \text{fail}.

```
  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 ]
```

### 5.3.25 HamiltonianPath

\[ \text{HamiltonianPath}(\text{digraph}) \]

**Returns:** A list or \text{fail}.

Returns a Hamiltonian path if one exists, \text{fail} if not.

A Hamiltonian path of a digraph with \( n \) vertices is directed cycle of length \( n \). If \( \text{digraph} \) is a digraph that contains a Hamiltonian path, then this function returns one, described in the form used by \( \text{DigraphAllSimpleCircuits} \) (5.3.20). Note if \( \text{digraph} \) has 0 or 1 vertices, then \( \text{HamiltonianPath} \) returns \([\] \) or \([1] \), respectively.

```
  gap> gr := DigraphSymmetricClosure(ChainDigraph(5));;
  gap> HamiltonianPath(gr);
  [ 5, 4, 3, 2, 1 ]
  gap> gr := CompleteDigraph(5);
  gap> HamiltonianPath(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> HamiltonianPath(gr);
  [ 1, 6, 5, 2, 4, 3 ]
```
The method used in this attribute has the same worst case complexity as DigraphMonomorphism (7.3.4).

Example

```gap
gap> g := Digraph([[]]);
<digraph with 1 vertex, 0 edges>
gap> HamiltonianPath(g);
[ 1 ]
gap> g := Digraph([[2], [1]]);
<digraph with 2 vertices, 2 edges>
gap> HamiltonianPath(g);
[ 1, 2 ]
gap> g := Digraph([[3], [], [2]]);
<digraph with 3 vertices, 2 edges>
gap> HamiltonianPath(g);
fail
gap> g := Digraph([[2], [3], [1]]);
<digraph with 3 vertices, 3 edges>
gap> HamiltonianPath(g);
[ 1, 2, 3 ]
```

5.4 Cayley graphs of groups

5.4.1 GroupOfCayleyDigraph

```gap
gap> G := DihedralGroup(IsPermGroup, 8);
Group([ (1,2,3,4), (2,4) ])
gap> digraph := CayleyDigraph(G);
<digraph with 8 vertices, 16 edges>
gap> GroupOfCayleyDigraph(digraph) = G;
true
```

5.4.2 GeneratorsOfCayleyDigraph

```gap
gap> G := DihedralGroup(IsPermGroup, 8);
Group([ (1,2,3,4), (2,4) ])
gap> digraph := CayleyDigraph(G);
<digraph with 8 vertices, 16 edges>
gap> GeneratorsOfCayleyDigraph(digraph)
[ (1,2,3,4), (2,4) ]
true
```

Returns: A list of generators.

If digraph is a Cayley graph of a group or semigroup with respect to a set of generators gens and digraph belongs to the category IsCayleyDigraph (3.1.2), then GeneratorsOfCayleyDigraph return the list of generators gens over which digraph is defined.

See also GroupOfCayleyDigraph (5.4.1) or SemigroupOfCayleyDigraph (5.4.1).
Example

```gap
gap> G := DihedralGroup(IsPermGroup, 8);
Group([ (1,2,3,4), (2,4) ])
gap> digraph := CayleyDigraph(G);
<digraph with 8 vertices, 16 edges>
gap> GeneratorsOfCayleyDigraph(digraph) = GeneratorsOfGroup(G);
true
gap> digraph := CayleyDigraph(G, []);
<digraph with 8 vertices, 8 edges>
gap> GeneratorsOfCayleyDigraph(digraph) = [];
true
```

5.5 Associated semigroups

5.5.1 AsSemigroup

▷ AsSemigroup(filt, digraph) 
▷ AsMonoid(filt, digraph) 

Returns: A semilattice of partial perms.

The operation AsSemigroup requires that filt be equal to IsPartialPermSemigroup (Reference: IsPartialPermSemigroup). If digraph is a IsJoinSemilatticeDigraph (6.1.15) or IsLatticeDigraph (6.1.15) then AsSemigroup returns a semigroup of partial perms which is isomorphic to the semigroup whose elements are the vertices of digraph with the binary operation PartialOrderDigraphJoinOfVertices (5.2.15). If digraph satisfies IsMeetSemilatticeDigraph (6.1.15) but not IsJoinSemilatticeDigraph (6.1.15) then AsSemigroup returns a semigroup of partial perms which is isomorphic to the semigroup whose elements are the vertices of digraph with the binary operation PartialOrderDigraphMeetOfVertices (5.2.15).

The operation AsMonoid behaves similarly to AsSemigroup except that filt may also be equal to IsPartialPermMonoid (Reference: IsPartialPermMonoid), digraph must satisfy IsLatticeDigraph (6.1.15), and the output satisfies IsMonoid (Reference: IsMonoid).

The output of both of these operations is guaranteed to be of minimal degree (see DegreeOfPartialPermSemigroup (Reference: DegreeOfPartialPermSemigroup)). Furthermore the GeneratorsOfSemigroup (Reference: GeneratorsOfSemigroup) of the output is guaranteed to be the unique generating set of minimal size.

Example

```gap
dl := Digraph([[1], [1, 2], [1, 3], [1, 4], [1, 2, 3, 5]]);
<digraph with 5 vertices, 11 edges>
s := AsSemigroup(IsPartialPermSemigroup, dl);
<partial perm semigroup of rank 3 with 4 generators>
ForAll(Elements(s), IsIdempotent);
true
IsInverseSemigroup(s);
true
Size(s);
5
dl := Digraph([[1], [1, 2], [1, 2, 3]]);
<digraph with 3 vertices, 6 edges>
M := AsMonoid(IsPartialPermMonoid, dl);
```
5.6 Planarity

5.6.1 KuratowskiPlanarSubdigraph

\[ \triangledown \text{KuratowskiPlanarSubdigraph}(\text{digraph}) \]

\textbf{Returns:} A list or fail.

KuratowskiPlanarSubdigraph returns the list of out-neighbours of a (not necessarily induced) subdigraph of the digraph \texttt{digraph} that witnesses the fact that \texttt{digraph} is not planar, or fail if \texttt{digraph} is planar. In other words, KuratowskiPlanarSubdigraph returns the out-neighbours of a subdigraph of \texttt{digraph} that is homeomorphic to the complete graph with 5 vertices, or to the complete bipartite graph with vertex sets of sizes 3 and 3.

The directions and multiplicities of any edges in \texttt{digraph} are ignored when considering whether or not \texttt{digraph} is planar.

See also \texttt{IsPlanarDigraph (6.4.1)} and \texttt{SubdigraphHomeomorphicToK33 (5.6.5)}.

This method uses the reference implementation in \texttt{edge-addition-planarity-suite} by John Boyer of the algorithms described in [BM06].

\begin{verbatim}
gap> D := Digraph([[3, 5, 10], [8, 9, 10], [1, 4], [3, 6], [1, 7, 11], [4, 7], [6, 8], [2, 7], [2, 11], [1, 2], [5, 9]]); <digraph with 11 vertices, 25 edges>
gap> KuratowskiPlanarSubdigraph(D); fail
\end{verbatim}

\begin{verbatim}
gap> D := Digraph([[2, 4, 7, 9, 10], [1, 3, 4, 6, 9, 10], [6, 10], [2, 5, 8, 9], [1, 2, 3, 4, 6, 7, 9, 10], [3, 4, 5, 7, 9, 10], [3, 4, 5, 6, 9, 10], [3, 4, 5, 7, 9], [2, 3, 5, 6, 7, 8], [3, 5]]); <digraph with 10 vertices, 50 edges>
gap> IsPlanarDigraph(D); false
gap> KuratowskiPlanarSubdigraph(D); [ 2, 9, 7 ], [ 3 ], [ 6 ], [ 5, 9 ], [ 6 ], [ ], [ 4 ], [ 7, 9, 3 ], [ ], [ ]
\end{verbatim}

5.6.2 KuratowskiOuterPlanarSubdigraph

\[ \triangledown \text{KuratowskiOuterPlanarSubdigraph}(\text{digraph}) \]

\textbf{Returns:} A list or fail.

KuratowskiOuterPlanarSubdigraph returns the list of out-neighbours of a (not necessarily induced) subdigraph of the digraph \texttt{digraph} that witnesses the fact that \texttt{digraph} is not outer planar, or fail if \texttt{digraph} is outer planar. In other words, KuratowskiOuterPlanarSubdigraph returns the out-neighbours of a subdigraph of \texttt{digraph} that is homeomorphic to the complete graph with 4 vertices, or to the complete bipartite graph with vertex sets of sizes 2 and 3.

The directions and multiplicities of any edges in \texttt{digraph} are ignored when considering whether or not \texttt{digraph} is outer planar.
See also IsOuterPlanarDigraph (6.4.2), SubdigraphHomeomorphicToK4 (5.6.5), and SubdigraphHomeomorphicToK23 (5.6.5).

This method uses the reference implementation in edge-addition-planarity-suite by John Boyer of the algorithms described in [BM06].

Example

```gap
d := Digraph([[3, 5, 10], [8, 9, 10], [1, 4], [3, 6], [1, 7, 11], [4, 7], [6, 8], [2, 7], [2, 11], [1, 2], [5, 9]]);
> <digraph with 11 vertices, 25 edges>
gap> KuratowskiOuterPlanarSubdigraph(d);
[ [ 3, 5, 10 ], [ 9, 8, 10 ], [ 4 ], [ 6 ], [ 11 ], [ 7 ], [ 8 ], [ ], [ 11 ], [ ], [ ] ]

d := Digraph([[2, 4, 7, 9, 10], [1, 3, 4, 6, 9, 10], [6, 10], [2, 5, 8, 9], [1, 2, 3, 4, 6, 7, 9, 10], [3, 4, 5, 7, 9, 10], [3, 4, 5, 6, 9, 10], [3, 4, 5, 7, 9, 10], [2, 3, 5, 6, 7, 8], [3, 5]]);
> <digraph with 10 vertices, 50 edges>
gap> IsOuterPlanarDigraph(d);
false

gap> KuratowskiOuterPlanarSubdigraph(d);
[ [ ], [ ], [ ], [ 9, 4 ], [ ], [ ], [ 9, 4 ], [ 7, 9 ], [ ] ]
```

5.6.3 PlanarEmbedding

▷ PlanarEmbedding(digraph) (attribute)

Returns: A list or fail.

If digraph is a planar digraph, then PlanarEmbedding returns the list of out-neighbours of a subdigraph of digraph such that each vertex’s neighbours are given in clockwise order. If digraph is not planar, then fail is returned.

The directions and multiplicities of any edges in digraph are ignored by PlanarEmbedding.

See also IsPlanarDigraph (6.4.1).

This method uses the reference implementation in edge-addition-planarity-suite by John Boyer of the algorithms described in [BM06].

Example

```gap
d := Digraph([[3, 5, 10], [8, 9, 10], [1, 4], [3, 6], [1, 7, 11], [4, 7], [6, 8], [2, 7], [2, 11], [1, 2], [5, 9]]);
> <digraph with 11 vertices, 25 edges>
gap> PlanarEmbedding(d);
[ [ 3, 10, 5 ], [ 10, 8, 9 ], [ 4 ], [ 6 ], [ 11, 7 ], [ 7 ], [ 8 ], [ ], [ 11 ], [ ], [ ] ]

d := Digraph([[2, 4, 7, 9, 10], [1, 3, 4, 6, 9, 10], [6, 10], [2, 5, 8, 9], [1, 2, 3, 4, 6, 7, 9, 10], [3, 4, 5, 7, 9, 10], [3, 4, 5, 6, 9, 10], [3, 4, 5, 7, 9, 10], [2, 3, 5, 6, 7, 8], [3, 5]]);
> <digraph with 10 vertices, 50 edges>
gap> PlanarEmbedding(d);
fail
```

5.6.4 OuterPlanarEmbedding

▷ OuterPlanarEmbedding(digraph) (attribute)

Returns: A list or fail.
If \( \text{digraph} \) is an outer planar digraph, then \( \text{OuterPlanarEmbedding} \) returns the list of out-neighbours of a subdigraph of \( \text{digraph} \) such that each vertex’s neighbours are given in clockwise order. If \( \text{digraph} \) is not outer planar, then \( \text{fail} \) is returned.

The directions and multiplicities of any edges in \( \text{digraph} \) are ignored by \( \text{OuterPlanarEmbedding} \).

See also \( \text{IsOuterPlanarDigraph} \) (6.4.2).

This method uses the reference implementation in edge-addition-planarity-suite by John Boyer of the algorithms described in [BM06].

**Example**

\[
\text{gap> D := Digraph([[3, 5, 10], [8, 9, 10], [1, 4], [3, 6],}
> [1, 7, 11], [4, 7], [6, 8], [2, 7], [2, 11], [1, 2], [5, 9]]);}
<digraph with 11 vertices, 25 edges>
\text{gap> OuterPlanarEmbedding(D);} fail
\text{gap> D := Digraph([[2, 4, 7, 9, 10], [1, 3, 4, 6, 9, 10], [6, 10],}
> [2, 5, 8, 9], [1, 2, 3, 4, 6, 7, 9, 10], [3, 4, 5, 7, 9, 10],
> [3, 4, 5, 6, 9, 10], [3, 4, 5, 7, 9], [2, 3, 5, 6, 7, 8], [3, 5]]);
<digraph with 10 vertices, 50 edges>
\text{gap> OuterPlanarEmbedding(D);} fail
\text{gap> OuterPlanarEmbedding(CompleteBipartiteDigraph(2, 2));}
[[ 3, 4 ], [ 4, 3 ], [ ], [ ]]
\]

### 5.6.5 SubdigraphHomeomorphicToK23

- SubdigraphHomeomorphicToK23(digraph) (attribute)
- SubdigraphHomeomorphicToK33(digraph) (attribute)
- SubdigraphHomeomorphicToK4(digraph) (attribute)

**Returns:** A list or \( \text{fail} \).

These attributes return the list of out-neighbours of a subdigraph of the digraph \( \text{digraph} \) which is homeomorphic to one of the following: the complete bipartite graph with vertex sets of sizes 2 and 3; the complete bipartite graph with vertex sets of sizes 3 and 3; or the complete graph with 4 vertices. If \( \text{digraph} \) has no such subgraphs, then \( \text{fail} \) is returned.

See also \( \text{IsPlanarDigraph} \) (6.4.1) and \( \text{IsOuterPlanarDigraph} \) (6.4.2) for more details.

This method uses the reference implementation in edge-addition-planarity-suite by John Boyer of the algorithms described in [BM06].

**Example**

\[
\text{gap> D := Digraph([[3, 5, 10], [8, 9, 10], [1, 4], [3, 6], [1, 7, 11],}
> [4, 7], [6, 8], [2, 7], [2, 11], [1, 2], [5, 9]]);}
<digraph with 11 vertices, 25 edges>
\text{gap> SubdigraphHomeomorphicToK4(D);} [ [ 3, 5, 10 ], [ 9, 8, 10 ], [ 4 ], [ 6 ], [ 7, 11 ], [ 7 ], [ 8 ],
[ ], [ 11 ], [ ], [ ] ]
\text{gap> SubdigraphHomeomorphicToK23(D);} [ [ 3, 5, 10 ], [ 9, 8, 10 ], [ 4 ], [ 6 ], [ 11 ], [ 7 ], [ 8 ],
[ ], [ 11 ], [ ], [ ] ]
\text{gap> D := Digraph([[3, 5, 10], [8, 9, 10], [1, 4], [3, 6], [1, 11],}
> [4, 7], [6, 8], [2, 7], [2, 11], [1, 2], [5, 9]]);}
<digraph with 11 vertices, 24 edges>
\text{gap> SubdigraphHomeomorphicToK4(D);}
Digraphs

fail

gap> SubdigraphHomeomorphicToK23(D);
[ [ 3, 10, 5 ], [ 10, 8, 9 ], [ 4 ], [ 6 ], [ 11 ], [ 7 ], [ 8 ],
  [ ], [ 11 ], [ ], [ ] ]
gap> SubdigraphHomeomorphicToK33(D);
fail
gap> SubdigraphHomeomorphicToK23(NullDigraph(0));
fail
gap> SubdigraphHomeomorphicToK33(CompleteDigraph(5));
fail
gap> SubdigraphHomeomorphicToK33(CompleteBipartiteDigraph(3, 3));
[ [ 4, 6, 5 ], [ 4, 5, 6 ], [ 6, 5, 4 ], [ ], [ ], [ ] ]
gap> SubdigraphHomeomorphicToK4(CompleteDigraph(3));
fail
Chapter 6

Properties of digraphs

6.1 Edge properties

6.1.1 DigraphHasLoops

\[ \text{DigraphHasLoops}(\text{digraph}) \]

\text{Returns:} \ true \ or \ false.

This property returns true if the digraph \text{digraph} has loops, and false if it does not. A loop is an edge with equal source and range.

\[ \text{Example} \]

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

6.1.2 IsAntisymmetricDigraph

\[ \text{IsAntisymmetricDigraph}(\text{digraph}) \]

\text{Returns:} \ true \ or \ false.

This property returns true if the digraph \text{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.

\[ \text{Example} \]

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

74

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 DigraphGreedyColouring (7.3.14).

See also DigraphBicomponents (5.3.12).

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 IsCompleteBipartiteDigraph

▷ IsCompleteBipartiteDigraph(digraph)

(property)

Returns: true or false.

Returns true if the digraph digraph is a complete bipartite digraph, and false if it is not. A digraph is a complete bipartite digraph if it is bipartite, see IsBipartiteDigraph (6.1.3), and there exists a unique edge with source i and range j if and only if i and j lie in different bicomponents of digraph, see DigraphBicomponents (5.3.12).

Equivalently, a bipartite digraph with bicomponents of size m and n is complete precisely when it has 2mn edges, none of which are multiple edges.

See also CompleteBipartiteDigraph (3.5.3).

Example

gap> gr := CycleDigraph(2);
<digraph with 2 vertices, 2 edges>
gap> IsCompleteBipartiteDigraph(gr);
true
gap> gr := CycleDigraph(4);
<digraph with 4 vertices, 4 edges>
gap> IsBipartiteDigraph(gr);
true
gap> IsCompleteBipartiteDigraph(gr);
false
6.1.5 **IsCompleteDigraph**

\[ \text{IsCompleteDigraph(digraph)} \]

- **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.

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

6.1.6 **IsEmptyDigraph**

- **IsEmptyDigraph(digraph)**
- **IsNullDigraph(digraph)**

  - **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.

```gap
gap> gr := Digraph([[], []]);
gap> IsEmptyDigraph(gr);
true
gap> IsNullDigraph(gr);
true
gap> gr := Digraph([[], [1]]);
gap> IsEmptyDigraph(gr);
false
gap> IsNullDigraph(gr);
false
```

6.1.7 **IsFunctionalDigraph**

- **IsFunctionalDigraph(digraph)**

  - **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.

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

6.1.8 IsMultiDigraph

▷ IsMultiDigraph(digraph)

(property)

Returns: true or false.

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

gap> gr := Digraph(["a", "b", "c"], ["a", "b", "b"], ["b", "c", "a"]);
<digraph with 3 vertices, 3 edges>
gap> IsMultiDigraph(gr);
false
gap> gr := DigraphFromDigraph6String("&Bug");
<digraph with 3 vertices, 6 edges>
gap> IsDuplicateFree(DigraphEdges(gr));
true
gap> IsMultiDigraph(gr);
true

6.1.9 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.

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

gap> gr := Digraph([[3, 1], [4, 2], [3], [2, 1]]);
<digraph with 4 vertices, 7 edges>
gap> IsReflexiveDigraph(gr);
false
6.1.10 \textbf{IsSymmetricDigraph}

\begin{verbatim}
\texttt{IsSymmetricDigraph(digraph)} \\
\texttt{(property)}
\end{verbatim}

\textbf{Returns: true or false.}

This property is true if the digraph \texttt{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, a symmetric digraph has a symmetric adjacency matrix \texttt{AdjacencyMatrix(5.2.1)}.

\begin{verbatim}
\texttt{Example}
\end{verbatim}

\begin{verbatim}
gap> gr1 := Digraph([[2], [1, 3], [2, 3]]); 
<digraph with 3 vertices, 5 edges>
gap> IsSymmetricDigraph(gr1); 
true
\end{verbatim}

\begin{verbatim}
gap> adj1 := AdjacencyMatrix(gr1);;
gap> Display(adj1); 
\[
\begin{bmatrix}
0 & 1 & 0 \\
1 & 0 & 1 \\
0 & 1 & 1 \\
\end{bmatrix}
\]
gap> adj1 = TransposedMat(adj1); 
true
\end{verbatim}

\begin{verbatim}
gap> gr1 = DigraphReverse(gr1); 
true
\end{verbatim}

\begin{verbatim}
gap> gr2 := Digraph([[2, 3], [1, 3], [2, 3]]); 
<digraph with 3 vertices, 6 edges>
gap> IsSymmetricDigraph(gr2); 
false
\end{verbatim}

\begin{verbatim}
gap> adj2 := AdjacencyMatrix(gr2);;
gap> Display(adj2); 
\[
\begin{bmatrix}
0 & 1 & 1 \\
1 & 0 & 1 \\
0 & 1 & 1 \\
\end{bmatrix}
\]
gap> adj2 = TransposedMat(adj2); 
false
\end{verbatim}

6.1.11 \textbf{IsTournament}

\begin{verbatim}
\texttt{IsTournament(digraph)} \\
\texttt{(property)}
\end{verbatim}

\textbf{Returns: true or false.}

This property is true if the digraph \texttt{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.

\begin{verbatim}
\texttt{Example}
\end{verbatim}

\begin{verbatim}
gap> gr := Digraph([[2, 3, 4], [3, 4], [4], []]); 
<digraph with 4 vertices, 6 edges>
gap> IsTournament(gr); 
true
\end{verbatim}

\begin{verbatim}
gap> gr := Digraph([[2], [1], [3]]); 
<digraph with 3 vertices, 3 edges>
\end{verbatim}
6.1.12 IsTransitiveDigraph

\texttt{gap> IsTransitiveDigraph(gr);}
\texttt{false}

\begin{itemize}
\item \textbf{IsTransitiveDigraph(digraph)} \hspace{1cm} (property)
\item \textbf{Returns:} true or false.
\end{itemize}

This property is true if the digraph \texttt{digraph} is transitive, and false if it is not. A digraph is \textit{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 \texttt{DigraphTopologicalSort (5.1.7)}], then methods with complexity \(O(m + n + m \cdot n)\) will be used when appropriate.

\begin{itemize}
\item \texttt{Example}
\item \texttt{gap> gr := Digraph([[1, 2], [3], [3]]);}
\item \texttt{<digraph with 3 vertices, 4 edges>}
\item \texttt{gap> IsTransitiveDigraph(gr);}
\item \texttt{false}
\item \texttt{gap> gr2 := Digraph([[1, 2, 3], [3], [3]]);}
\item \texttt{<digraph with 3 vertices, 5 edges>}
\item \texttt{gap> IsTransitiveDigraph(gr2);}
\item \texttt{true}
\item \texttt{gap> gr2 = DigraphTransitiveClosure(gr);}
\item \texttt{true}
\item \texttt{gap> gr3 := Digraph([[1, 2, 2, 3], [3, 3], [3]]);}
\item \texttt{<multidigraph with 3 vertices, 7 edges>}
\item \texttt{gap> IsTransitiveDigraph(gr3);}
\item \texttt{true}
\end{itemize}

6.1.13 IsPreorderDigraph

\texttt{gap> gr := Digraph([[1, 2, 2, 3], [3, 3], [3]]);}
\texttt{<digraph with 3 vertices, 5 edges>}
\texttt{gap> IsPreorderDigraph(gr);}
\texttt{true}
\texttt{gap> gr := Digraph([[1 .. 4], [1 .. 4], [1 .. 4], [1 .. 4]]);}
\texttt{<digraph with 4 vertices, 16 edges>}
\texttt{gap> IsPreorderDigraph(gr);}
\texttt{true}
\texttt{gap> gr := Digraph([[2], [3], [4], [5], [1]]);}

\begin{itemize}
\item \textbf{IsPreorderDigraph(digraph)} \hspace{1cm} (property)
\item \textbf{IsQuasiorderDigraph(digraph)} \hspace{1cm} (property)
\item \textbf{Returns:} true or false.
\end{itemize}

A digraph is a preorder digraph if and only if the digraph satisfies both \texttt{IsReflexiveDigraph (6.1.9)} and \texttt{IsTransitiveDigraph (6.1.12)}. A preorder digraph (or quasiorder digraph) \texttt{digraph} corresponds to the preorder relation \(\leq\) defined by \(x \leq y\) if and only if \([x, y]\) is an edge of \texttt{digraph}.

\begin{itemize}
\item \texttt{Example}
\item \texttt{gap> gr := Digraph([[1, 2, 3], [2, 3]]);}
\item \texttt{<digraph with 3 vertices, 5 edges>}
\item \texttt{gap> IsPreorderDigraph(gr);}
\item \texttt{true}
\item \texttt{gap> gr := Digraph([[1 .. 4], [1 .. 4], [1 .. 4], [1 .. 4]]);}
\item \texttt{<digraph with 4 vertices, 16 edges>}
\item \texttt{gap> IsPreorderDigraph(gr);}
\item \texttt{true}
\item \texttt{gap> gr := Digraph([[2], [3], [4], [5], [1]]);}
\end{itemize}
6.1.14 IsPartialOrderDigraph

▷ IsPartialOrderDigraph(digraph) (property)

Returns: true or false.

A digraph is a partial order digraph if and only if the digraph satisfies all of IsReflexiveDigraph (6.1.9), IsAntisymmetricDigraph (6.1.2) and IsTransitiveDigraph (6.1.12). A partial order digraph corresponds to the partial order relation \( \leq \) defined by \( x \leq y \) if and only if \([x, y]\) is an edge of digraph.

Example

\[
\begin{align*}
\text{gap> gr := Digraph([[1], [1, 2], [2, 3]]);} \\
\text{<digraph with 3 vertices, 5 edges>} \\
\text{gap> IsPartialOrderDigraph(gr);} \\
\text{true} \\
\text{gap> gr := CycleDigraph(5);} \\
\text{<digraph with 5 vertices, 5 edges>} \\
\text{gap> IsPartialOrderDigraph(gr);} \\
\text{false} \\
\text{gap> gr := Digraph([[1, 3], [2, 3], [3]]);} \\
\text{<digraph with 3 vertices, 5 edges>} \\
\text{gap> IsPartialOrderDigraph(gr);} \\
\text{true}
\end{align*}
\]

6.1.15 IsMeetSemilatticeDigraph

▷ IsMeetSemilatticeDigraph(digraph) (property)  
▷ IsJoinSemilatticeDigraph(digraph) (property)  
▷ IsLatticeDigraph(digraph) (property)

Returns: true or false.

IsMeetSemilatticeDigraph returns true if the digraph digraph is a meet semilattice; IsJoinSemilatticeDigraph returns true if the digraph digraph is a join semilattice; and IsLatticeDigraph returns true if the digraph digraph is both a meet and a join semilattice.

For a partial order digraph IsPartialOrderDigraph (6.1.14) the corresponding partial order is the relation \( \leq \), defined by \( x \leq y \) if and only if \([x, y]\) is an edge. A digraph is a meet semilattice if it is a partial order and every pair of vertices has a greatest lower bound (meet) with respect to the aforementioned relation. A join semilattice is a partial order where every pair of vertices has a least upper bound (join) with respect to the relation.

Example

\[
\begin{align*}
\text{gap> gr := Digraph([[1, 3], [2, 3], [3]]);} \\
\text{<digraph with 3 vertices, 5 edges>} \\
\text{gap> IsMeetSemilatticeDigraph(gr);} \\
\text{false}
\end{align*}
\]
6.2 Regularity

6.2.1 IsInRegularDigraph

[Property]

\[ \text{IsInRegularDigraph}(\text{digraph}) \]

**Returns:** true or false.

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

[Example]

\[
\text{gap> } \text{IsInRegularDigraph(ColoredGraph(4))}; \\
\text{true} \\
\text{gap> } \text{IsInRegularDigraph(ColoredGraph(5))}; \\
\text{false}
\]

6.2.2 IsOutRegularDigraph

[Property]

\[ \text{IsOutRegularDigraph}(\text{digraph}) \]

**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
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
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
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

\begin{verbatim}
> IsAcyclicDigraph(digraph)

Returns: true or false.
This property is true if the digraph \textit{digraph} is acyclic, and false if it is not. A digraph is acyclic if every directed cycle on the digraph is trivial. See section 1.1.1 for the definition of a directed cycle, and of a trivial directed cycle.

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.
\end{verbatim}

Example

\begin{verbatim}
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 := DigraphFromDiSparse6String(
> "\QCGIDBaPGkU6i6KQhCeRIdhrHu6ZfRdDAbPhT1|zF");
<digraph with 35 vertices, 34 edges>
gap> IsAcyclicDigraph(gr); // true

gap> IsAcyclicDigraph(ChainDigraph(10)); // true

gap> IsAcyclicDigraph(CycleDigraph(10)); // false
\end{verbatim}

6.3.2 IsChainDigraph

\begin{verbatim}
> IsChainDigraph(digraph)

Returns: true or false.
IsChainDigraph returns true if the digraph \textit{digraph} is isomorphic to the chain digraph with the same number of vertices as \textit{digraph}, and false if it is not; see ChainDigraph (3.5.1).

A digraph is a chain if and only if it is a directed tree, in which every vertex has out degree at most one; see IsDirectedTree (6.3.7) and OutDegrees (5.2.8).
\end{verbatim}

Example

\begin{verbatim}
gap> gr := Digraph([[1, 3], [2, 3], [3]]);
<digraph with 3 vertices, 5 edges>
gap> IsChainDigraph(gr); // false

gap> gr := ChainDigraph(5);
<digraph with 5 vertices, 4 edges>
gap> IsChainDigraph(gr); // true

gap> gr := DigraphReverse(gr);
<digraph with 5 vertices, 4 edges>
gap> IsChainDigraph(gr); // true
\end{verbatim}
6.3.3 IsConnectedDigraph

\[ \text{IsConnectedDigraph}(\text{digraph}) \] (property)

**Returns:** true or false.

This property is true if the digraph \( \text{digraph} \) is weakly connected and false if it is not. A digraph \( \text{digraph} \) 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 (5.2.5) 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).

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

6.3.4 IsBiconnectedDigraph

\[ \text{IsBiconnectedDigraph}(\text{digraph}) \] (property)

**Returns:** true or false.

A connected digraph is biconnected if it is still connected (in the sense of IsConnectedDigraph (6.3.3)) when any vertex is removed. IsBiconnectedDigraph returns true if the digraph \( \text{digraph} \) is biconnected, and false if it is not. In particular, IsBiconnectedDigraph returns false if \( \text{digraph} \) is not connected.

Multiple edges and loops are ignored by this method.

The method used in this operation has complexity \( O(m + n) \) where \( m \) is the number of edges (counting multiple edges as one, and not counting loops) and \( n \) is the number of vertices in the digraph. See also ArticulationPoints (5.3.13).

```gap
gap> IsBiconnectedDigraph(Digraph([[1, 3], [2, 3], [3]]));
false
gap> IsBiconnectedDigraph(CycleDigraph(5));
true
gap> digraph := Digraph([[1, 1], [1, 1, 2], [3], [3, 3, 4, 4]]);
gap> IsBiconnectedDigraph(digraph);
false
```

6.3.5 IsStronglyConnectedDigraph

\[ \text{IsStronglyConnectedDigraph}(\text{digraph}) \] (property)

**Returns:** true or false.

This property is true if the digraph \( \text{digraph} \) is strongly connected and false if it is not.

A digraph \( \text{digraph} \) is strongly connected if there is a directed path from every vertex to every other vertex. See section 1.1.1 for the definition of a directed path.

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.

```gap
gap> IsStronglyConnectedDigraph(Digraph([[1, 3], [2, 3], [3]]));
false
gap> IsStronglyConnectedDigraph(CycleDigraph(5));
true
gap> digraph := Digraph([[1, 1], [1, 1, 2], [3], [3, 3, 4, 4]]);
gap> IsStronglyConnectedDigraph(digraph);
false
```
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
```

### 6.3.6 IsAperiodicDigraph

**IsAperiodicDigraph**(digraph) (property)

Returns: true or false.

This property is true if the digraph *digraph* is aperiodic, i.e. if its *DigraphPeriod* (5.3.14) 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
```

### 6.3.7 IsDirectedTree

**IsDirectedTree**(digraph) (property)

Returns: true or false.

Returns true if the digraph *digraph* is a directed tree, and false if it is not.

A directed tree is an acyclic digraph with precisely 1 source, such that no two vertices share an out-neighbour. Note the empty digraph is not considered a directed tree as it has no source.

See also *DigraphSources* (5.1.6).

Example

```gap
gr := Digraph([[], [2]]);
<digraph with 2 vertices, 1 edge>
gap> IsDirectedTree(gr);
false

gap> gr := Digraph([[3], [3], []]);
<digraph with 3 vertices, 2 edges>
gap> IsDirectedTree(gr);
false

gap> gr := Digraph([[2], [3], []];
<digraph with 3 vertices, 2 edges>
gap> IsDirectedTree(gr);
true

gap> gr := Digraph([[2, 3], [6], [4, 5], [], [], []]);
<digraph with 6 vertices, 5 edges>
```
6.3.8 IsUndirectedTree

- **IsUndirectedTree**(digraph)
- **IsUndirectedForest**(digraph)

**Returns:** true or false.

The property *IsUndirectedTree* returns true if the digraph *digraph* is an undirected tree, and the property *IsUndirectedForest* returns true if *digraph* is an undirected forest; otherwise, these properties return false.

An undirected tree is a symmetric digraph without loops, in which for any pair of distinct vertices *u* and *v*, there is exactly one directed path from *u* to *v*. See *IsSymmetricDigraph* (6.1.10) and *DigraphHasLoops* (6.1.1), and see section 1.1.1 for the definition of directed path. This definition implies that an undirected tree has no multiple edges.

An undirected forest is a digraph, each of whose connected components is an undirected tree. In other words, an undirected forest is isomorphic to a disjoint union of undirected trees. See *DigraphConnectedComponents* (5.3.8) and *DigraphDisjointUnion* (3.3.26). In particular, every undirected tree is an undirected forest.

Please note that the digraph with zero vertices is considered to be neither an undirected tree nor an undirected forest.

**Example**

```gap
gap> gr := Digraph([[3], [3], [1, 2]]);  # digraph with 3 vertices, 4 edges
<digraph with 3 vertices, 4 edges>
gap> IsUndirectedTree(gr);               # true
true
gap> IsSymmetricDigraph(gr) and not DigraphHasLoops(gr); # true
true
gap> gr := Digraph([[3], [5], [1, 4], [3], [2]]);  # digraph with 5 vertices, 6 edges
<digraph with 5 vertices, 6 edges>
gap> IsConnectedDigraph(gr);             # false
false
gap> IsUndirectedTree(gr);              # false
false
gap> IsUndirectedForest(gr);            # true
true
gap> gr := Digraph([[1, 2], [1], [2]]);  # digraph with 3 vertices, 4 edges
<digraph with 3 vertices, 4 edges>
gap> IsUndirectedTree(gr) or IsUndirectedForest(gr); # false
false
gap> IsSymmetricDigraph(gr) or not DigraphHasLoops(gr); # false
false
```

6.3.9 IsEulerianDigraph

- **IsEulerianDigraph**(digraph)

**Returns:** true or false.

This property returns true if the digraph *digraph* is Eulerian.
A digraph is called Eulerian if there exists a directed circuit on the digraph which includes every edge exactly once. See section 1.1.1 for the definition of a directed circuit.

```
gap> gr := Digraph([[1]]);
<digraph with 1 vertex, 0 edges>
gap> IsEulerianDigraph(gr);
true
gap> gr := Digraph([[2], [1]]);
<digraph with 2 vertices, 1 edge>
gap> IsEulerianDigraph(gr);
false
gap> gr := Digraph([[3], [], [2]]);
<digraph with 3 vertices, 2 edges>
gap> IsEulerianDigraph(gr);
false
gap> gr := Digraph([[2], [3], [1]]);
<digraph with 3 vertices, 3 edges>
gap> IsEulerianDigraph(gr);
true
```

6.3.10 IsHamiltonianDigraph

\[ \text{IsHamiltonianDigraph}(\text{digraph}) \]  \hspace{1cm} (property)

**Returns:** true or false.

If \text{digraph} is Hamiltonian, then this property returns true, and false if it is not.

A digraph with \( n \) vertices is Hamiltonian if it has a directed cycle of length \( n \). See Section 1.1.1 for the definition of a directed cycle. Note the empty digraphs on 0 and 1 vertices are considered to be Hamiltonian.

The method used in this operation has the worst case complexity as \text{DigraphMonomorphism} (7.3.4).

```
gap> g := Digraph([[1]]);
<digraph with 1 vertex, 0 edges>
gap> IsHamiltonianDigraph(g);
true
gap> g := Digraph([[2], [1]]);
<digraph with 2 vertices, 2 edges>
gap> IsHamiltonianDigraph(g);
true
gap> g := Digraph([[3], [], [2]]);
<digraph with 3 vertices, 2 edges>
gap> IsHamiltonianDigraph(g);
false
gap> g := Digraph([[2], [3], [1]]);
<digraph with 3 vertices, 3 edges>
gap> IsHamiltonianDigraph(g);
true
```
6.3.11 IsCycleDigraph

▷ IsCycleDigraph(digraph) (property)

**Returns:** true or false.

IsCycleDigraph returns true if the digraph `digraph` is isomorphic to the cycle digraph with the same number of vertices as `digraph`, and false if it is not; see CycleDigraph (3.5.5).

A digraph is a *cycle* if and only if it is strongly connected and has the same number of edges as vertices.

Example

```gap
gap> gr := Digraph([[1, 3], [2, 3], [3]]);
<digraph with 3 vertices, 5 edges>
gap> IsCycleDigraph(gr);
false
gap> gr := CycleDigraph(5);
<digraph with 5 vertices, 5 edges>
gap> IsCycleDigraph(gr);
true
gap> gr := OnDigraphs(gr, (1, 2, 3));
<digraph with 5 vertices, 5 edges>
gap> gr = CycleDigraph(5);
false
gap> IsCycleDigraph(gr);
true
```

6.4 Planarity

6.4.1 IsPlanarDigraph

▷ IsPlanarDigraph(digraph) (property)

**Returns:** true or false.

A *planar* digraph is a digraph that can be embedded in the plane in such a way that its edges do not intersect. A digraph is planar if and only if it does not have a subdigraph that is homeomorphic to either the complete graph on 5 vertices or the complete bipartite graph with vertex sets of sizes 3 and 3.

IsPlanarDigraph returns true if the digraph `digraph` is planar and false if it is not. The directions and multiplicities of any edges in `digraph` are ignored by IsPlanarDigraph.

See also IsOuterPlanarDigraph (6.4.2).

This method uses the reference implementation in edge-addition-planarity-suite by John Boyer of the algorithms described in [BM06].

Example

```gap
gap> IsPlanarDigraph(CompleteDigraph(4));
true
gap> IsPlanarDigraph(CompleteDigraph(5));
false
gap> IsPlanarDigraph(CompleteBipartiteDigraph(2, 3));
true
gap> IsPlanarDigraph(CompleteBipartiteDigraph(3, 3));
false
```
6.4.2 IsOuterPlanarDigraph

\textbf{IsOuterPlanarDigraph}(\texttt{digraph}) \hfill (property)

\textbf{Returns:} true or false.

An outer planar digraph is a digraph that can be embedded in the plane in such a way that its edges do not intersect, and all vertices belong to the unbounded face of the embedding. A digraph is outer planar if and only if it does not have a subdigraph that is homeomorphic to either the complete graph on 4 vertices or the complete bipartite graph with vertex sets of sizes 2 and 3.

\texttt{IsOuterPlanarDigraph} returns true if the digraph \texttt{digraph} is outer planar and false if it is not. The directions and multiplicities of any edges in \texttt{digraph} are ignored by \texttt{IsPlanarDigraph}.

See also \texttt{IsPlanarDigraph} (6.4.1). This method uses the reference implementation in \texttt{edge-addition-planarity-suite} by John Boyer of the algorithms described in [BM06].

\begin{verbatim}
gap> IsOuterPlanarDigraph(CompleteDigraph(4)); false
gap> IsOuterPlanarDigraph(CompleteDigraph(5)); false
gap> IsOuterPlanarDigraph(CompleteBipartiteDigraph(2, 3)); false
gap> IsOuterPlanarDigraph(CompleteBipartiteDigraph(3, 3)); false
gap> IsOuterPlanarDigraph(CycleDigraph(10)); true
\end{verbatim}
Chapter 7

Homomorphisms

7.1 Acting on digraphs

7.1.1 OnDigraphs (for a digraph and a perm)

\[ \text{OnDigraphs}(\text{digraph}, \text{perm}) \]

\[ \text{OnDigraphs}(\text{digraph}, \text{trans}) \]

\textbf{Returns:} A digraph.

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

If the second argument is a transformation \textit{trans} of the vertices of \textit{digraph}, then this operation returns a digraph constructed by transforming the source and range of each edge according to \textit{trans}. Thus a vertex which does not appear in the image of \textit{trans} will be isolated in the returned digraph, and the returned digraph may contain multiple edges, even if \textit{digraph} does not. If \textit{trans} is mathematically a permutation, then the result coincides with \( \text{OnDigraphs}(\text{digraph}, \text{AsPermutation}(\text{trans})) \).

The \textbf{DigraphVertexLabels} (5.1.9) of \textit{digraph} will not be retained in the returned digraph.

\begin{example}
\begin{verbatim}
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);
<multidigraph with 3 vertices, 2 edges>
gap> OutNeighbours(new);
[ [ 2, 2 ], [ ], [ ] ]
gap> ForAll(DigraphEdges(gr),
true
\end{verbatim}
\end{example}
7.1.2 OnMultiDigraphs

\[ \text{OnMultiDigraphs} \text{(digraph, pair)} \] (operation)

\[ \text{OnMultiDigraphs} \text{(digraph, perm1, perm2)} \] (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 := BlissCanonicalLabelling(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

From version 0.11.0 of `Digraphs` it is possible to use either bliss or nauty (via NautyTracesInterface) to calculate canonical labellings and automorphism groups of digraphs; see [JK07] and [MP14] for more details about bliss and nauty, respectively.

7.2.1 DigraphsUseNauty

\[ \text{DigraphsUseNauty()} \] (function)

\[ \text{DigraphsUseBliss()} \] (function)

**Returns:** Nothing.

These functions can be used to specify whether nauty or bliss should be used by default by Digraphs. If NautyTracesInterface is not available, then these functions do nothing. Otherwise, by calling `DigraphsUseNauty` subsequent computations will default to using nauty rather than bliss, where possible.

You can call these functions at any point in a GAP session, as many times as you like, it is guaranteed that existing digraphs remain valid, and that comparison of existing digraphs and newly created digraphs via `IsIsomorphicDigraph` (7.2.14), `IsIsomorphicDigraph` (7.2.15), `IsomorphismDigraphs` (7.2.16), and `IsomorphismDigraphs` (7.2.17) are also valid.

It is also possible to compute the automorphism group of a specific digraph using both nauty and bliss using `NautyAutomorphismGroup` (7.2.4) and `BlissAutomorphismGroup` (7.2.3), respectively.
7.2.2 AutomorphismGroup (for a digraph)

\[ \text{AutomorphismGroup}(\text{digraph}) \]

**Returns:** A permutation group.

If \( \text{digraph} \) is a digraph, then this attribute contains the group of automorphisms of \( \text{digraph} \). An automorphism of \( \text{digraph} \) is an isomorphism from \( \text{digraph} \) to itself. See IsomorphismDigraphs (7.2.16) for more information about isomorphisms of digraphs.

The form in which the automorphism group is returned depends on whether \( \text{digraph} \) has multiple edges; see IsMultiDigraph (6.1.8).

**for a digraph without multiple edges**

If \( \text{digraph} \) has no multiple edges, then the automorphism group is returned as a group of permutations on the vertices of \( \text{digraph} \).

**for a multidigraph**

If \( \text{digraph} \) is a multidigraph, then the automorphism group is a group of permutations on the vertices and edges of \( \text{digraph} \).

For convenience, the group is returned as the direct product \( G \) of the group of automorphisms of the vertices of \( \text{digraph} \) with the stabiliser of the vertices in the automorphism group of the edges. These two groups can be accessed using the operation Projection (Reference: Projection for a domain and a positive integer), with the second argument being 1 or 2, respectively.

The permutations in the group Projection\((G, 1)\) act on the vertices of \( \text{digraph} \), and the permutations in the group Projection\((G, 2)\) act on the indices of DigraphEdges\((\text{digraph})\).

By default, the automorphism group is found using bliss by Tommi Junttila and Petteri Kaski. If NautyTracesInterface is available, then nauty by Brendan McKay and Adolfo Piperno can be used instead; see BlissAutomorphismGroup (7.2.3), NautyAutomorphismGroup (7.2.4), DigraphsUseBliss (7.2.1), and DigraphsUseNauty (7.2.1).

---

**Example**

```gap
gap> johnson := DigraphFromGraph6String("E}lw");
<digraph with 6 vertices, 24 edges>
gap> G := AutomorphismGroup(johnson);
Group([ (3,4), (2,3)(4,5), (1,2)(5,6) ])
gap> cycle := CycleDigraph(9);
<digraph with 9 vertices, 9 edges>
gap> G := AutomorphismGroup(cycle);
Group([ (1,2,3,4,5,6,7,8,9) ])
gap> IsCyclic(G) and Size(G) = 9;
true
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
```
Digraphs

```
gap> gr := Digraph([[2], [3, 3], [3], [2]]);
<multidigraph with 4 vertices, 5 edges>
gap> G := AutomorphismGroup(gr);
Group([ (1,2), (3,4) ])
gap> P1 := Projection(G, 1);
1st projection of Group([ (1,2), (3,4) ])
gap> P2 := Projection(G, 2);
2nd projection of Group([ (1,2), (3,4) ])
gap> DigraphVertices(gr);
[ 1 .. 4 ]
gap> Range(P1);
Group([ (1,4) ])
gap> DigraphEdges(gr);
[ [ 1, 2 ], [ 2, 3 ], [ 2, 3 ], [ 3, 3 ], [ 4, 2 ] ]
gap> Range(P2);
Group([ (2,3) ])```

7.2.3 BlissAutomorphismGroup

▷ BlissAutomorphismGroup(digraph[, colours])

*Returns*: A permutation group.

If `digraph` is a digraph, then this attribute contains the group of automorphisms of `digraph` as calculated using bliss by Tommi Junttila and Petteri Kaski.

The attribute `AutomorphismGroup` (7.2.2) and operation `AutomorphismGroup` (7.2.5) returns the value of either `BlissAutomorphismGroup` or `NautyAutomorphismGroup` (7.2.4). These groups are, of course, equal but their generating sets may differ.

See also `DigraphsUseBliss` (7.2.1), and `DigraphsUseNauty` (7.2.1).

```
gap> BlissAutomorphismGroup(JohnsonDigraph(5, 2));
Group([ (3,4)(6,7)(8,9), (2,3)(5,6)(9,10), (2,5)(3,6)(4,7), (1,2)(6,8)(7,9) ])```

7.2.4 NautyAutomorphismGroup

▷ NautyAutomorphismGroup(digraph[, colours])

*Returns*: A permutation group.

If `digraph` is a digraph, then this attribute contains the group of automorphisms of `digraph` as calculated using nauty by Brendan McKay and Adolfo Piperno via NautyTracesInterface. The attribute `AutomorphismGroup` (7.2.2) and operation `AutomorphismGroup` (7.2.5) returns the value of either `NautyAutomorphismGroup` or `BlissAutomorphismGroup` (7.2.3). These groups are, of course, equal but their generating sets may differ.

See also `DigraphsUseBliss` (7.2.1), and `DigraphsUseNauty` (7.2.1).

```
gap> NautyAutomorphismGroup(JohnsonDigraph(5, 2));
Group([ (3,4)(6,7)(8,9), (2,3)(5,6)(9,10), (2,5)(3,6)(4,7), (1,2)(6,8)(7,9) ])```
7.2.5 AutomorphismGroup (for a digraph and a homogeneous list)

\[ \text{AutomorphismGroup}(\text{digraph}, \text{colours}) \]

(\text{operation})

\textbf{Returns:} A permutation group.

This operation computes the automorphism group of a coloured digraph. A coloured digraph can be specified by its underlying digraph \texttt{digraph} and its colouring \texttt{colours}. Let \( n \) be the number of vertices of \texttt{digraph}. The colouring \texttt{colours} may have one of the following two forms:

- a list of \( n \) integers, where \texttt{colours}[i] is the colour of vertex \( i \), using the colours \([1 \ldots m]\) for some \( m \leq n \); or

- a list of non-empty disjoint lists whose union is \texttt{DigraphVertices(digraph)}, such that \texttt{colours}[i] is the list of all vertices with colour \( i \).

The \textit{automorphism group} of a coloured digraph \texttt{digraph} with colouring \texttt{colours} is the group consisting of its automorphisms; an \textit{automorphism} of \texttt{digraph} is an isomorphism of coloured digraphs from \texttt{digraph} to itself. This group is equal to the subgroup of \texttt{AutomorphismGroup(digraph)} consisting of those automorphisms that preserve the colouring specified by \texttt{colours}. See \texttt{AutomorphismGroup (7.2.2)}, and see \texttt{IsomorphismDigraphs (7.2.17)} for more information about isomorphisms of coloured digraphs.

The form in which the automorphism group is returned depends on whether \texttt{digraph} has multiple edges; see \texttt{IsMultiDigraph (6.1.8)}.

\textbf{for a digraph without multiple edges}

If \texttt{digraph} has no multiple edges, then the automorphism group is returned as a group of permutations on the vertices of \texttt{digraph}.

\textbf{for a multidigraph}

If \texttt{digraph} is a multidigraph, then the automorphism group is a group of permutations on the vertices and edges of \texttt{digraph}.

For convenience, the group is returned as the direct product \( G \) of the group of automorphisms of the vertices of \texttt{digraph} with the stabiliser of the vertices in the automorphism group of the edges. These two groups can be accessed using the operation \texttt{Projection (Reference: Projection for a domain and a positive integer)}, with the second argument being 1 or 2, respectively.

The permutations in the group \texttt{Projection(G, 1)} act on the vertices of \texttt{digraph}, and the permutations in the group \texttt{Projection(G, 2)} act on the indices of \texttt{DigraphEdges(digraph)}.

By default, the automorphism group is found using \texttt{bliss} by Tommi Junttila and Petteri Kaski. If \texttt{NautyTracesInterface} is available, then \texttt{nauty} by Brendan McKay and Adolfo Piperno can be used instead; see \texttt{BlissAutomorphismGroup (7.2.3)}, \texttt{NautyAutomorphismGroup (7.2.4)}, \texttt{DigraphsUseBliss (7.2.1)}, and \texttt{DigraphsUseNauty (7.2.1)}.

\begin{verbatim}
Example
\end{verbatim}

\begin{verbatim}
gap> cycle := CycleDigraph(9);<digraph with 9 vertices, 9 edges>
gap> G := AutomorphismGroup(cycle);;
gap> IsCyclic(G) and Size(G) = 9; true
gap> colours := [[1, 4, 7], [2, 5, 8], [3, 6, 9]];
gap> H := AutomorphismGroup(cycle, colours);;
\end{verbatim}
\begin{verbatim}
gap> Size(H);
3
gap> H = AutomorphismGroup(cycle, [1, 2, 3, 1, 2, 3, 1, 2, 3]);
true
gap> H = SubgroupByProperty(G, p -> OnTuplesSets(colours, p) = colours);
true
gap> IsTrivial(AutomorphismGroup(cycle, [1, 1, 2, 2, 2, 2, 2, 2, 2]));
true
gap> gr := Digraph([[2], [3, 3], [3], [2], [2]]);
<multidigraph with 5 vertices, 6 edges>

gap> G := AutomorphismGroup(gr, [1, 1, 2, 3, 1]);
Group([ (1,2), (3,4) ])

gap> P1 := Projection(G, 1);
1st projection of Group([ (1,2), (3,4) ])

gap> P2 := Projection(G, 2);
2nd projection of Group([ (1,2), (3,4) ])

gap> DigraphVertices(gr);
[ 1 .. 5 ]

gap> Range(P1);
Group([ (1,5) ])

gap> DigraphEdges(gr);
[ [ 1, 2 ], [ 2, 3 ], [ 2, 3 ], [ 2, 3 ], [ 4, 2 ], [ 5, 2 ] ]

gap> Range(P2);
Group([ (2,3) ])
\end{verbatim}

### 7.2.6 BlissCanonicalLabelling (for a digraph)

- **BlissCanonicalLabelling** *(digraph)*
- **NautyCanonicalLabelling** *(digraph)*

**Returns:** A permutation, or a list of two permutations.

A function \( \rho \) that maps a digraph to a digraph is a **canonical representative map** if the following two conditions hold for all digraphs \( G \) and \( H \):

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

A **canonical labelling** of a digraph \( G \) (under \( \rho \)) is an isomorphism of \( G \) onto its **canonical representative**, \( \rho(G) \). See IsomorphismDigraphs (7.2.16) for more information about isomorphisms of digraphs.

BlissCanonicalLabelling returns a canonical labelling of the digraph **digraph** found using bliss by Tommi Junttila and Petteri Kaski. NautyCanonicalLabelling returns a canonical labelling of the digraph **digraph** found using nauty by Brendan McKay and Adolfo Piperno. Note that the canonical labellings returned by bliss and nauty are not usually the same (and may depend of the version used).

The form of the canonical labelling returned by BlissCanonicalLabelling depends on whether **digraph** has multiple edges; see IsMultiDigraph (6.1.8).

**for a digraph without multiple edges**

If the digraph **digraph** has no multiple edges, then the canonical labelling of **digraph** is given
as a permutation of its vertices. The canonical representative of \textit{digraph} can be created from \textit{digraph} and its canonical labelling \textit{p} by using the operation \texttt{OnDigraphs} (7.1.1):

\begin{verbatim}
gap> OnDigraphs(digraph, p);
\end{verbatim}

for a multidigraph

The canonical labelling of the multidigraph \textit{digraph} is given as a pair \textit{P} of permutations. The first, \textit{P[1]}, is a permutation of the vertices of \textit{digraph}. The second, \textit{P[2]}, is a permutation of the edges of \textit{digraph}; it acts on the indices of the list \texttt{DigraphEdges(digraph)}. The canonical representative of \textit{digraph} can be created from \textit{digraph} and its canonical labelling \textit{P} by using the operation \texttt{OnMultiDigraphs} (7.1.2):

\begin{verbatim}
gap> OnMultiDigraphs(digraph, P);
\end{verbatim}

\begin{verbatim}
gap> digraph1 := DigraphFromDiSparse6String(".ImNS_AiB?qRN");
<digraph with 10 vertices, 8 edges>
gap> BlissCanonicalLabelling(digraph1);
(1,3,4)(2,10,6,7,9,8)
gap> p := (1, 2, 7, 5)(3, 9)(6, 10, 8);;
gap> digraph2 := OnDigraphs(digraph1, p);
<digraph with 10 vertices, 8 edges>
gap> digraph1 = digraph2;
false
\end{verbatim}

\begin{verbatim}
gap> gr := DigraphFromDiSparse6String(".ImEk|O@SK?od");
<multidigraph with 10 vertices, 10 edges>
gap> BlissCanonicalLabelling(gr);
[ (1,9,7,5)(2,10,3), (1,6,9)(2,5,10,4,8)(3,7) ]
gap> gr := Digraph([[2], [3, 3], [3], [2], [2]]);
<multidigraph with 5 vertices, 6 edges>
gap> BlissCanonicalLabelling(gr, [1, 2, 2, 1, 3]);
[ (1,2,4), (1,2,6,4,3,5) ]
\end{verbatim}

7.2.7 BlissCanonicalLabelling (for a digraph and a list)

\begin{itemize}
\item \texttt{BlissCanonicalLabelling(digraph, colours)} (operation)
\item \texttt{NautyCanonicalLabelling(digraph, colours)} (operation)
\end{itemize}

\textbf{Returns:} A permutation.

A function \(\rho\) that maps a coloured digraph to a coloured digraph is a \textit{canonical representative map} if the following two conditions hold for all coloured digraphs \(G\) and \(H\):

\begin{itemize}
\item \(\rho(G)\) and \(G\) are isomorphic as coloured digraphs; and
\item \(\rho(G) = \rho(H)\) if and only if \(G\) and \(H\) are isomorphic as coloured digraphs.
\end{itemize}

A \textit{canonical labelling} of a coloured digraph \(G\) (under \(\rho\)) is an isomorphism of \(G\) onto its \textit{canonical representative}, \(\rho(G)\). See \texttt{IsomorphismDigraphs} (7.2.17) for more information about isomorphisms of coloured digraphs.
A coloured digraph can be specified by its underlying digraph \textit{digraph} and its colouring \textit{colours}. Let \( n \) be the number of vertices of \textit{digraph}. The colouring \textit{colours} may have one of the following two forms:

- a list of \( n \) integers, where \( \text{colours}[i] \) is the colour of vertex \( i \), using the colours \([1 \ldots m]\) for some \( m \leq n \); or
- a list of non-empty disjoint lists whose union is \textit{DigraphVertices(digraph)}, such that \( \text{colours}[i] \) is the list of all vertices with colour \( i \).

If \textit{digraph} and \textit{colours} together form a coloured digraph, BlissCanonicalLabelling returns a canonical labelling of the digraph \textit{digraph} found using bliss by Tommi Junttila and Petteri Kaski. Similarly, NautyCanonicalLabelling returns a canonical labelling of the digraph \textit{digraph} found using nauty by Brendan McKay and Adolfo Piperno. Note that the canonical labellings returned by bliss and nauty are not usually the same (and may depend on the version used).

The form of the canonical labelling returned by BlissCanonicalLabelling depends on whether \textit{digraph} has multiple edges; see IsMultiDigraph (6.1.8).

\textbf{for a digraph without multiple edges}

If the digraph \textit{digraph} has no multiple edges, then the canonical labelling of \textit{digraph} is given as a permutation of its vertices. The canonical representative of \textit{digraph} can be created from \textit{digraph} and its canonical labelling \( p \) by using the operation OnDigraphs (7.1.1):

\begin{verbatim}
gap> OnDigraphs(digraph, p);\end{verbatim}

\textbf{for a multidigraph}

The canonical labelling of the multidigraph \textit{digraph} is given as a pair \( P \) of permutations. The first, \( P[1] \), is a permutation of the vertices of \textit{digraph}. The second, \( P[2] \), is a permutation of the edges of \textit{digraph}; it acts on the indices of the list \textit{DigraphEdges(digraph)}. The canonical representative of \textit{digraph} can be created from \textit{digraph} and its canonical labelling \( P \) by using the operation OnMultiDigraphs (7.1.2):

\begin{verbatim}
gap> OnMultiDigraphs(digraph, P);\end{verbatim}

In either case, the colouring of the canonical representative can easily be constructed. A vertex \( v \) (in \textit{digraph}) has colour \( i \) if and only if the vertex \( v^p \) (in the canonical representative) has colour \( i \), where \( p \) is the permutation of the canonical labelling that acts on the vertices of \textit{digraph}. In particular, if \textit{colours} has the first form that is described above, then the colouring of the canonical representative is given by:

\begin{verbatim}
gap> List(DigraphVertices(digraph), i -> colours[i / p]);\end{verbatim}

On the other hand, if \textit{colours} has the second form above, then the canonical representative has colouring:

\begin{verbatim}
gap> OnTuplesSets(colours, p);\end{verbatim}
Example

```
gap> digraph := DigraphFromDiSparse6String(".ImNS_AiB?qRN");
<digraph with 10 vertices, 8 edges>
gap> colours := [[1, 2, 8, 9, 10], [3, 4, 5, 6, 7]];;
gap> p := BlissCanonicalLabelling(digraph, colours);
(2,3,7,10)(4,6,9,5,8)
gap> OnDigraphs(digraph, p);
<digraph with 10 vertices, 8 edges>
gap> OnTuplesSets(colours, p);
[ [ 1, 2, 3, 4, 5 ], [ 6, 7, 8, 9, 10 ] ]
```

7.2.8 BlissCanonicalDigraph

\[\text{BlissCanonicalDigraph}(\text{digraph})\]

\[\text{NautyCanonicalDigraph}(\text{digraph})\]

Returns: A digraph.

The attribute BlissCanonicalLabelling returns the canonical representative found by applying BlissCanonicalLabelling (7.2.6). The digraph returned is canonical in the sense that

- BlissCanonicalDigraph(digraph) and digraph are isomorphic as digraphs; and

- If gr is any digraph then BlissCanonicalDigraph(gr) and BlissCanonicalDigraph(digraph) are equal if and only if gr and digraph are isomorphic as digraphs.

Analogously, the attribute NautyCanonicalLabelling returns the canonical representative found by applying NautyCanonicalLabelling (7.2.6).

Example

```
```

7.2.9 DigraphGroup

\[\text{DigraphGroup}(\text{digraph})\]

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 its automorphism group, then DigraphGroup returns the entire automorphism group of digraph.
Note that certain other constructor operations such as CayleyDigraph (3.1.10), BipartiteDoubleDigraph (3.3.32), and DoubleDigraph (3.3.31), 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
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
    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
    Group([ (1,2,3,4)(5,6,7,8), (1,5)(2,6)(3,7)(4,8) ])
gap> AutomorphismGroup(ddigraph) =
    Group([ (2,4)(3,5)(6,8), (1,2)(3,4)(5,6)(7,8) ]);  
    true
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) ])  
    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
    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) ])  
```
7.2.10 DigraphOrbits

\[ \text{DigraphOrbits}(\text{digraph}) \]

\textbf{Returns: } A list of lists of integers.

\text{DigraphOrbits} returns the orbits of the action of the \text{DigraphGroup} (7.2.9) on the set of vertices of \text{digraph}.

\begin{example}
\begin{verbatim}
gap> G := Group([(2, 3)(7, 8, 9), (1, 2, 3)(4, 5, 6)(8, 9)]);
gap> gr := EdgeOrbitsDigraph(G, [1, 2]);
<digraph with 9 vertices, 6 edges>
gap> DigraphOrbits(gr);
\end{verbatim}
\end{example}

7.2.11 DigraphOrbitReps

\[ \text{DigraphOrbitReps}(\text{digraph}) \]

\textbf{Returns: } A list of integers.

\text{DigraphOrbitReps} returns a list of orbit representatives of the action of the \text{DigraphGroup} (7.2.9) on the set of vertices of \text{digraph}.

\begin{example}
\begin{verbatim}
gap> digraph := CayleyDigraph(AlternatingGroup(4));
<digraph with 12 vertices, 24 edges>
gap> DigraphOrbitReps(digraph);
[ 1 ]
gap> digraph := DigraphFromDigraph6String("&IGO??S?'?_@?a?CK?O");
<digraph with 10 vertices, 14 edges>
gap> DigraphOrbitReps(digraph);
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
\end{verbatim}
\end{example}

7.2.12 DigraphSchreierVector

\[ \text{DigraphSchreierVector}(\text{digraph}) \]

\textbf{Returns: } A list of integers.

\text{DigraphSchreierVector} returns the so-called \textit{Schreier vector} of the action of the \text{DigraphGroup} (7.2.9) on the set of vertices of \text{digraph}. The Schreier vector is a list \text{sch} of integers with length \text{DigraphNrVertices(digraph)} where:

\begin{enumerate}
  \item \text{sch}[i] < 0:
    \begin{itemize}
      \item implies that \text{i} is an orbit representative and \text{DigraphOrbitReps(digraph)}[-\text{sch}[i]] = \text{i}.
    \end{itemize}
  \item \text{sch}[i] > 0:
    \begin{itemize}
      \item implies that \text{i} / \text{gens[\text{sch}[i]]} is one step closer to the root (or representative) of the tree, where \text{gens} is the generators of \text{DigraphGroup(digraph)}.
    \end{itemize}
\end{enumerate}

\begin{example}
\begin{verbatim}
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 ]
\end{verbatim}
\end{example}
digraphs

```gap
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
```

7.2.13 DigraphStabilizer

▷ DigraphStabilizer(digraph, v)

**Returns:** A permutation group.

DigraphStabilizer returns the stabilizer of the vertex v under of the action of the DigraphGroup (7.2.9) on the set of vertices of digraph.

```gap
gap> digraph := DigraphFromDigraph6String("&GYHPQgWTIIPW");<digraph with 8 vertices, 24 edges>
gap> DigraphStabilizer(digraph, 8);
Group(())
gap> DigraphStabilizer(digraph, 2);
Group(())
```

7.2.14 IsIsomorphicDigraph (for digraphs)

▷ IsIsomorphicDigraph(digraph1, digraph2)

**Returns:** true or false.

This operation returns true if there exists an isomorphism from the digraph digraph1 to the digraph digraph2. See IsomorphismDigraphs (7.2.16) for more information about isomorphisms of digraphs.

By default, an isomorphism is found using the canonical labellings of the digraphs obtained from bliss by Tommi Junttila and Petteri Kaski. If NautyTracesInterface is available, then nauty by Brendan Mckay and Adolfo Piperno can be used instead; see DigraphsUseBliss (7.2.1), and DigraphsUseNauty (7.2.1).

```gap
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 := DigraphFromDiSparse6String(".IiGdqrHiogeaF");
<multidigraph with 10 vertices, 10 edges>
gap> digraph2 := DigraphFromDiSparse6String(".IiK'K@FFSouF_|^");
<multidigraph with 10 vertices, 10 edges>
gap> IsIsomorphicDigraph(digraph1, digraph2);
```
Digraphs

false

\texttt{gap> digraph1 := Digraph([[3], [], []]);}
\texttt{<digraph with 3 vertices, 1 edge>}
\texttt{gap> digraph2 := Digraph([[], [], [2]]);}
\texttt{<digraph with 3 vertices, 1 edge>}
\texttt{gap> IsIsomorphicDigraph(digraph1, digraph2);}
\texttt{true}

7.2.15 \textbf{IsIsomorphicDigraph (for digraphs and homogeneous lists)}

\texttt{\texttt{\textbackslash \textbackslash \texttt{operation}}}

\textbf{Returns:} true or false.

This operation tests for isomorphism of coloured digraphs. A coloured digraph can be specified by its underlying digraph \texttt{digraph1} and its colouring \texttt{colours1}. Let \(n\) be the number of vertices of \texttt{digraph1}. The colouring \texttt{colours1} may have one of the following two forms:

- a list of \(n\) integers, where \texttt{colours\[i\]} is the colour of vertex \(i\), using the colours \([1 \ldots m]\) for some \(m <= n\); or
- a list of non-empty disjoint lists whose union is \texttt{DigraphVertices(digraph)}, such that \texttt{colours\[i\]} is the list of all vertices with colour \(i\).

If \texttt{digraph1} and \texttt{digraph2} are digraphs without multiple edges, and \texttt{colours1} and \texttt{colours2} are colourings of \texttt{digraph1} and \texttt{digraph2}, respectively, then this operation returns \texttt{true} if there exists an isomorphism between these two coloured digraphs. See \texttt{IsomorphismDigraphs (7.2.17)} for more information about isomorphisms of coloured digraphs.

By default, an isomorphism is found using the canonical labellings of the digraphs obtained from \texttt{bliss} by Tommi Junttila and Petteri Kaski. If \texttt{NautyTracesInterface} is available, then \texttt{nauty} by Brendan McKay and Adolfo Piperno can be used instead; see \texttt{DigraphsUseBliss (7.2.1)}, and \texttt{DigraphsUseNauty (7.2.1)}.

\texttt{\texttt{\textbackslash \textbackslash \texttt{Example}}}

\texttt{\texttt{gap> digraph1 := ChainDigraph(4);}}
\texttt{<digraph with 4 vertices, 3 edges>}
\texttt{gap> digraph2 := ChainDigraph(3);}
\texttt{<digraph with 3 vertices, 2 edges>}
\texttt{gap> IsIsomorphicDigraph(digraph1, digraph2,}
\texttt{ > [[1, 4], [2, 3]], [[1, 2], [3]]);}  \texttt{false}
\texttt{gap> digraph2 := DigraphReverse(digraph1);}
\texttt{<digraph with 4 vertices, 3 edges>}
\texttt{gap> IsIsomorphicDigraph(digraph1, digraph2,}
\texttt{ > [1, 1, 1, 1], [1, 1, 1, 1]);}
\texttt{true}
\texttt{gap> IsIsomorphicDigraph(digraph1, digraph2,}
\texttt{ > [1, 2, 2, 1], [1, 2, 2, 1]);}
\texttt{true}
\texttt{gap> IsIsomorphicDigraph(digraph1, digraph2,}
\texttt{ > [1, 1, 2, 2], [1, 1, 2, 2]);}
\texttt{false}
\texttt{gap> digraph1 := Digraph([[2, 1, 2], [1, 2, 1]]);}  \texttt{false}
\texttt{<multidigraph with 2 vertices, 6 edges>
7.2.16 IsomorphismDigraphs (for digraphs)

\[ \text{IsomorphismDigraphs}(\text{digraph1}, \text{digraph2}) \]

\textbf{Returns:} A permutation, or a pair of permutations, or fail.
This operation returns an isomorphism between the digraphs \text{digraph1} and \text{digraph2} if one exists, else this operation returns fail.

\textbf{for digraphs without multiple edges}

An isomorphism from a digraph \text{digraph1} to a digraph \text{digraph2} is a bijection \( p \) from the vertices of \text{digraph1} to the vertices of \text{digraph2} with the following property: for all vertices \( i \) and \( j \) of \text{digraph1}, \([i, j]\) is an edge of \text{digraph1} if and only if \([i \ ^{p}, \ j \ ^{p}]\) is an edge of \text{digraph2}.

If there exists such an isomorphism, then this operation returns one. The form of this isomorphism is a permutation \( p \) of the vertices of \text{digraph1} such that
\[ \text{OnDigraphs} \left( \text{digraph1}, \ p \right) = \text{digraph2}. \]

\textbf{for multidigraphs}

An isomorphism from a multidigraph \text{digraph1} to a multidigraph \text{digraph2} is a bijection \( P[1] \) from the vertices of \text{digraph1} to the vertices of \text{digraph2} and a bijection \( P[2] \) from the indices of edges of \text{digraph1} to the indices of edges of \text{digraph2} with the following property: \([i, j]\) is the \( k \)th edge of \text{digraph1} if and only if \([i \ ^{P[1]}, \ j \ ^{P[1]}]\) is the \((k \ ^{P[2]})\)th edge of \text{digraph2}.

If there exists such an isomorphism, then this operation returns one. The form of this isomorphism is a pair of permutations \( P \) — where the first is a permutation of the vertices of \text{digraph1} and the second is a permutation of the indices of \text{DigraphEdges} \left( \text{digraph1} \right) — such that
\[ \text{OnMultiDigraphs} \left( \text{digraph1}, P \right) = \text{digraph2}. \]

By default, an isomorphism is found using the canonical labellings of the digraphs obtained from \text{bliss} by Tommi Junttila and Petteri Kaski. If \text{NautyTracesInterface} is available, then \text{nauty} by Brendan Mckay and Adolfo Piperno can be used instead; see \text{DigraphsUseBliss} \left( \text{7.2.1} \right), and \text{DigraphsUseNauty} \left( \text{7.2.1} \right).
Digraphs

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

\texttt{gap> OnDigraphs(digraph1, p) = digraph2;}
true
\texttt{gap> digraph1 := DigraphFromDiSparse6String(".ImNS_?DSE@ce[~");}
\texttt{<multidigraph with 10 vertices, 10 edges>}
\texttt{gap> digraph2 := DigraphFromDiSparse6String(".IkOlQefi_kg0f");}
\texttt{<multidigraph with 10 vertices, 10 edges>}
\texttt{gap> IsomorphismDigraphs(digraph1, digraph2);
[ (1,9,5,3,10,6,4,7,2), (1,8,6,3,7)(2,9,4,10,5) ]}
\texttt{gap> digraph1 := DigraphByEdges([[7, 10], [7, 10]], 10);
<multidigraph with 10 vertices, 2 edges>}
\texttt{gap> digraph2 := DigraphByEdges([[2, 3], [2, 3]], 10);
<multidigraph with 10 vertices, 2 edges>}
\texttt{gap> IsomorphismDigraphs(digraph1, digraph2);
[ (2,4,6,8,9,10,3,5,7), () ]}

7.2.17 IsomorphismDigraphs (for digraphs and homogeneous lists)

\texttt{\textbf{\texttt{>>> IsomorphismDigraphs(digraph1, digraph2, colours1, colours2)}}}

(\textit{operation})

\textbf{Returns:} A permutation, or \texttt{fail}.

This operation searches for an isomorphism between coloured digraphs. A coloured digraph can be specified by its underlying digraph \texttt{digraph1} and its colouring \texttt{colours1}. Let \texttt{n} be the number of vertices of \texttt{digraph1}. The colouring \texttt{colours1} may have one of the following two forms:

- a list of \texttt{n} integers, where \texttt{colours[i]} is the colour of vertex \texttt{i}, using the colours [1 .. m] for some \texttt{m <= n}; or
- a list of non-empty disjoint lists whose union is \texttt{DigraphVertices(digraph)}, such that \texttt{colours[i]} is the list of all vertices with colour \texttt{i}.

An isomorphism between coloured digraphs is an isomorphism between the underlying digraphs that preserves the colourings. See \texttt{IsomorphismDigraphs (7.2.16)} for more information about isomorphisms of digraphs. More precisely, let \texttt{f} be an isomorphism of digraphs from the digraph \texttt{digraph1} (with colouring \texttt{colours1}) to the digraph \texttt{digraph2} (with colouring \texttt{colours2}), and let \texttt{p} be the permutation of the vertices of \texttt{digraph1} that corresponds to \texttt{f}. Then \texttt{f} preserves the colourings of \texttt{digraph1} and \texttt{digraph2} – and hence is an isomorphism of coloured digraphs – if \texttt{colours1[i] = colours2[i ^ p]} for all vertices \texttt{i} in \texttt{digraph1}.

This operation returns such an isomorphism if one exists, else it returns \texttt{fail}.

By default, an isomorphism is found using the canonical labellings of the digraphs obtained from \texttt{bliss} by Tommi Junttila and Petteri Kaski. If \texttt{NautyTracesInterface} is available, then \texttt{nauty} by Brendan Mckay and Adolfo Piperno can be used instead; see \texttt{DigraphsUseBliss (7.2.1)}, and \texttt{DigraphsUseNauty (7.2.1)}.

\texttt{gap> digraph1 := ChainDigraph(4);
<digraph with 4 vertices, 3 edges>}
\texttt{gap> digraph2 := ChainDigraph(3);
<digraph with 3 vertices, 2 edges>}
\texttt{gap> IsomorphismDigraphs(digraph1, digraph2,
> \texttt{[[1, 4], [2, 3]], [[1, 2], [3]]});
fail}
digraph2 := DigraphReverse(digraph1);
<digraph with 4 vertices, 3 edges>
colours1 := [1, 1, 1, 1];;
colours2 := [1, 1, 1, 1];;
p := IsomorphismDigraphs(digraph1, digraph2, colours1, colours2);
(1,4)(2,3)
OnDigraphs(digraph1, p) = digraph2;
true
List(DigraphVertices(digraph1), i -> colours1[i ^ p]) = colours2;
true
colours1 := [1, 1, 2, 2];;
colours2 := [2, 2, 1, 1];;
p := IsomorphismDigraphs(digraph1, digraph2, colours1, colours2);
(1,4)(2,3)
OnDigraphs(digraph1, p) = digraph2;
true
List(DigraphVertices(digraph1), i -> colours1[i ^ p]) = colours2;
true
IsomorphismDigraphs(digraph1, digraph2,
> [1, 1, 2, 2], [1, 1, 2, 2]);
fail
digraph1 := Digraph([[2, 2], [2], [1]]);
<multidigraph with 3 vertices, 4 edges>
digraph2 := Digraph([[1], [1, 1], [2]]);
<multidigraph with 3 vertices, 4 edges>
IsomorphismDigraphs(digraph1, digraph2,
> [1, 2, 2], [2, 1, 2]);

7.2.18 RepresentativeOutNeighbours

This function returns the list out of out-neighbours of each representative of the orbits of the action of DigraphGroup (7.2.9) 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.

Example

gap> digraph := Digraph(;
> [ 2, 1, 3, 4, 5 ], [ 3, 5 ], [ 2 ], [ 1, 2, 3, 5 ], [ 1, 2, 3, 4 ]);
<digraph with 5 vertices, 16 edges>
digraphGroup(digraph);
Group(())
digraph := DigraphFromDigraph6String("&GYHPQgWTIIPW");
<digraph with 8 vertices, 24 edges>
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) ])

Digraphs

RepresentativeOutNeighbours(digraph)

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 (7.2.9) 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 (7.2.9) is trivial, then OutNeighbours (5.2.6) is returned.
7.2.19 **IsDigraphIsomorphism**

\[
\text{IsDigraphIsomorphism} \quad (\text{operation})
\]

\[
\text{IsDigraphAutomorphism} \quad (\text{operation})
\]

**Returns:** true or false.

IsDigraphIsomorphism returns true if the permutation or transformation \(x\) is an isomorphism from the digraph \(src\) to the digraph \(ran\).

IsDigraphAutomorphism returns true if the permutation or transformation \(x\) is an automorphism of the digraph \(digraph\).

A permutation or transformation \(x\) is an isomorphism from a digraph \(src\) to a digraph \(ran\) if the following hold:

- \(x\) is a bijection from the vertices of \(src\) to those of \(ran\);
- \([u \sim x, v \sim x]\) is an edge of \(ran\) if and only if \([u, v]\) is an edge of \(src\); and
- \(x\) fixes every \(i\) which is not a vertex of \(src\).

See also AutomorphismGroup (7.2.2).

For some digraphs, it can be faster to use IsDigraphAutomorphism than to test membership in the automorphism group of \(digraph\).

**Example**

\[
\text{Example}
\]

```
gap> src := Digraph([[1], [1, 2], [1, 3]]); <digraph with 3 vertices, 5 edges>
gap> IsDigraphAutomorphism(src, (1, 2, 3)); false
gap> IsDigraphAutomorphism(src, (2, 3)); true
gap> IsDigraphAutomorphism(src, (2, 3)(4, 5)); false
gap> IsDigraphAutomorphism(src, (1, 4)); false
gap> IsDigraphAutomorphism(src, ()); true
gap> ran := Digraph([[2, 1], [2], [2, 3]]); <digraph with 3 vertices, 5 edges>

gap> IsDigraphIsomorphism(src, ran, (1, 2)); true

gap> IsDigraphIsomorphism(ran, src, (1, 2)); true

gap> IsDigraphIsomorphism(ran, src, (1, 2)); true

gap> IsDigraphIsomorphism(src, Digraph([[3], [1, 3], [2]]), (1, 2, 3)); false
```
7.2.20 IsDigraphColouring

**IsDigraphColouring**: operation

\texttt{IsDigraphColouring(digraph, list)}

\texttt{IsDigraphColouring(digraph, t)}

**Returns**: true or false.

The operation IsDigraphColouring verifies whether or not the list list describes a proper colouring of the digraph digraph.

A list list describes a proper colouring of a digraph digraph if list consists of positive integers, the length of list equals the number of vertices in digraph, and for any vertices u, v of digraph if u and v are adjacent, then list[u] > list[v].

A transformation t describes a proper colouring of a digraph digraph, if ImageListOfTransformation(t, DigraphNrVertices(digraph)) is a proper colouring of digraph.

See also IsDigraphHomomorphism (7.3.10).

Example

```gap
gap> D := JohnsonDigraph(5, 3);
<digraph with 10 vertices, 60 edges>
gap> IsDigraphColouring(D, [1, 2, 3, 3, 2, 1, 4, 5, 6, 7]);
true
gap> IsDigraphColouring(D, [1, 2, 3, 3, 2, 1, 2, 5, 6, 7]);
false
gap> IsDigraphColouring(D, [1, 2, 3, 3, 2, 1, 2, 5, 6, -1]);
false
gap> IsDigraphColouring(D, [1, 2, 3]);
false
gap> IsDigraphColouring(D, IdentityTransformation);
true
```

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**: function

\texttt{HomomorphismDigraphsFinder(D1, D2, hook, user_param, max_results, hint, injective, image, partial_map, colors1, colors2[, order])}

**Returns**: The argument user_param.

This function finds homomorphisms from the digraph D1 to the digraph D2 subject to the conditions imposed by the other arguments as described below.

If f and g are homomorphisms found by HomomorphismDigraphsFinder, then f cannot be obtained from g by right multiplying by an automorphism of D2.

**hook**

This argument should be a function or fail.

If hook is a function, then it must 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 HomomorphismDigraphsFinder.
If `hook` is `fail`, then a default function is used which simply adds every new homomorphism found by `HomomorphismDigraphsFinder` to `user_param`, which must be a mutable 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 of 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 mutable list.

**max_results**

This argument should be a positive integer or `infinity`. `HomomorphismDigraphsFinder` will return after it has found `max_results` homomorphisms or the search is complete, whichever happens first.

**hint**

This argument should be a positive integer or `fail`.

If `hint` is a positive integer, then only homomorphisms 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 0, 1, or 2. If it is 2, then only embeddings are found, if it is 1, then only injective homomorphisms are found, and if it is 0 there are no restrictions imposed by this argument.

For backwards compatibility, `injective` can also be `false` or `true` which correspond to the values 0 and 1 described in the previous paragraph, respectively.

**image**

This argument should be a subset of the vertices of the graph `D2`. `HomomorphismDigraphsFinder` only finds homomorphisms from `D1` to the subgraph of `D2` induced by the vertices `image`.

**partial_map**

This argument should be a partial map from `D1` to `D2`, that is, a (not necessarily dense) list of vertices of the digraph `D2` of length no greater than the number vertices in the digraph `D1`. `HomomorphismDigraphsFinder` only finds homomorphisms extending `partial_map` (if any).

**colors1**

This should be a list representing possible colours of vertices in the digraph `D1`; see `AutomorphismGroup (7.2.5)` for details of the permissible values for this argument.

**colors2**

This should be a list representing possible colours of vertices in the digraph `D2`; see `AutomorphismGroup (7.2.5)` for details of the permissible values for this argument.

**order**

The optional final argument `order` specifies the order the vertices in `D1` appear in the search for...
Digraphs

homomorphisms. The value of this parameter can have a large impact on the runtime of the function. It seems in many cases to be a good idea for this to be the DigraphWelshPowellOrder (7.3.15), i.e. vertices ordered from highest to lowest degree.

Example

```gap
gap> D := Chain Digraph(10);
<digraph with 10 vertices, 9 edges>
gap> D := DigraphSymmetricClosure(D);
<digraph with 10 vertices, 18 edges>
gap> HomomorphismDigraphsFinder(D, D, fail, [], infinity, 2, 0,
  > [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> D2 := Complete Digraph(6);
<digraph with 6 vertices, 15 edges>
gap> HomomorphismDigraphsFinder(D, D2, fail, [], 1, fail, 0,
  > [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(D, D2, func, [Transformation([2, 2])],
    > 3, fail, 0, [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 ] ) ]
```

7.3.2DigraphHomomorphism

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 to adjacent vertices.

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

Example

```gap
gap> gr1 := Chain Digraph(3);
<digraph with 3 vertices, 2 edges>
gap> gr2 := Digraph([ [3, 5], [2], [3, 1], [], [4]]);
<digraph with 5 vertices, 6 edges>
gap> DigraphHomomorphism(gr1, gr1);
IdentityTransformation
gap> map := DigraphHomomorphism(gr1, gr2);
Transformation( [ 3, 1, 5, 4, 5 ] )
gap> IsDigraphHomomorphism(gr1, gr2, map);
true
```
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 (7.2.2) 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.

```gap
Example
```
```gap
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 ] ),
gap> HomomorphismsDigraphsRepresentatives(gr1, CompleteDigraph(3));
[ Transformation( [ 2, 1 ] ), Transformation( [ 2, 1, 2 ] ) ]
```

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.

```gap
Example
```
```gap
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( [ 3, 1, 5, 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), except they only return injective homomorphisms.

```gap
Example
```
```gap
gap> gr1 := ChainDigraph(3);;
gap> gr2 := Digraph([[3, 5], [2], [3, 1], [], [4]]);
```
Digraphs

<digraph with 5 vertices, 6 edges>

\[
\text{gap> MonomorphismsDigraphs(gr1, gr2);}
\text{[ Transformation( [ 1, 5, 4, 4, 5 ] ),}
\text{ Transformation( [ 3, 1, 5, 4, 5 ] ) ]}
\]

\[
\text{gap> MonomorphismsDigraphsRepresentatives(gr1, CompleteDigraph(3));}
\text{[ Transformation( [ 2, 1 ] ) ]}
\]

7.3.6 DigraphEpimorphism

\[
\text{DigraphEpimorphism(digraph1, digraph2)}
\]

\textbf{Returns:} A transformation, or \texttt{fail}.

DigraphEpimorphism returns a single \textit{surjective} DigraphHomomorphism (7.3.2) between \texttt{digraph1} and \texttt{digraph2} if one exists, otherwise it returns \texttt{fail}.

\textbf{Example}

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

7.3.7 EpimorphismsDigraphs

\[
\text{EpimorphismsDigraphs(digraph1, digraph2)}
\]

\textbf{Returns:} A list of transformations.

These operations behave the same as \texttt{HomomorphismsDigraphs} (7.3.3) and \texttt{HomomorphismsDigraphsRepresentatives} (7.3.3), except they only return \textit{surjective} homomorphisms.

\textbf{Example}

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

\[
\text{gap> EpimorphismsDigraphs(gr1, gr2);}
\]
7.3.8 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
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 ] )
```

7.3.9 EmbeddingsDigraphs

> EmbeddingsDigraphs(D1, D2)  
  (operation)

> EmbeddingsDigraphsRepresentatives(D1, D2)  
  (operation)

Returns: A list of transformations.

These operations behave the same as HomomorphismsDigraphs (7.3.3) and HomomorphismsDigraphsRepresentatives (7.3.3), except they only return embeddings of `D1` into `D2`.

See also IsDigraphEmbedding (7.3.11).

Example

```gap
gap> D1 := NullDigraph(2);
<digraph with 2 vertices, 0 edges>
gap> D2 := CycleDigraph(5);
<digraph with 5 vertices, 5 edges>
gap> EmbeddingsDigraphsRepresentatives(D1, D2);
[ Transformation( [ 1, 3, 3 ] ), Transformation( [ 1, 4, 3, 4 ] ) ]
gap> EmbeddingsDigraphs(D1, D2);
[ Transformation( [ 1, 3, 3 ] ), Transformation( [ 1, 4, 3, 4 ] ),
  Transformation( [ 2, 4, 4, 5, 1 ] ),
  Transformation( [ 2, 5, 4, 5, 1 ] ),
  Transformation( [ 3, 1, 5, 1, 2 ] ),
  Transformation( [ 3, 5, 5, 1, 2 ] ),
  Transformation( [ 4, 1, 1, 2, 3 ] ),
  Transformation( [ 4, 2, 1, 2, 3 ] ),
  Transformation( [ 5, 2, 2, 3, 4 ] ),
  Transformation( [ 5, 3, 2, 3, 4 ] ) ]
```

7.3.10 IsDigraphHomomorphism

> IsDigraphHomomorphism(src, ran, x)  
  (operation)

> IsDigraphEpimorphism(src, ran, x)  
  (operation)
Digraphs

\[ \text{IsDigraphHomomorphism}(src, ran, x) \]
\[ \text{IsDigraphEndomorphism}(digraph, x) \]

**Returns:** true or false.

IsDigraphHomomorphism returns true if the permutation or transformation \( x \) is a homomorphism from the digraph \( src \) to the digraph \( ran \).

IsDigraphEpimorphism returns true if the permutation or transformation \( x \) is a surjective homomorphism from the digraph \( src \) to the digraph \( ran \).

IsDigraphMonomorphism returns true if the permutation or transformation \( x \) is an injective homomorphism from the digraph \( src \) to the digraph \( ran \).

IsDigraphEndomorphism returns true if the permutation or transformation \( x \) is an endomorphism of the digraph \( digraph \).

A permutation or transformation \( x \) is a homomorphism from a digraph \( src \) to a digraph \( ran \) if the following hold:

- \([u ^ x, v ^ x]\) is an edge of \( ran \) whenever \([u, v]\) is an edge of \( src \); and
- \( x \) fixes every \( i \) which is not a vertex of \( src \).

See also GeneratorsOfEndomorphismMonoid (7.3.12).

Example

```gap
gap> src := Digraph([[1], [1, 2], [1, 3]]);  
<digraph with 3 vertices, 5 edges>  
gap> ran := Digraph([[1], [1, 2]]);  
<digraph with 2 vertices, 3 edges>  
gap> IsDigraphHomomorphism(src, ran, Transformation([1, 2, 2]));  
true  
gap> IsDigraphHomomorphism(src, ran, Transformation([2, 1, 2]));  
false  
gap> IsDigraphHomomorphism(src, ran, Transformation([3, 3, 3]));  
false  
gap> IsDigraphHomomorphism(src, src, Transformation([3, 3, 3]));  
true  
gap> IsDigraphEndomorphism(src, Transformation([3, 3, 3]));  
true  
gap> IsDigraphEpimorphism(src, ran, Transformation([3, 3, 3]));  
false  
gap> IsDigraphMonomorphism(src, ran, Transformation([1, 2, 2]));  
false  
gap> IsDigraphEpimorphism(src, ran, Transformation([1, 2, 2]));  
true  
gap> IsDigraphMonomorphism(ran, src, ());  
true
```

7.3.11 IsDigraphEmbedding

\[ \text{IsDigraphEmbedding}(src, ran, x) \]

**Returns:** true or false.

IsDigraphEmbedding returns true if the permutation or transformation \( x \) is an embedding of the digraph \( src \) into the digraph \( ran \).
A permutation or transformation \( x \) is a **embedding** of a digraph \( \text{src} \) into a digraph \( \text{ran} \) if \( x \) is a monomorphism from \( \text{src} \) to \( \text{ran} \) and the inverse of \( x \) is a monomorphism from the subdigraph of \( \text{ran} \) induced by the image of \( x \) to \( \text{src} \). See also \text{IsDigraphHomomorphism} (7.3.10).

Example

```gap
gap> src := Digraph([[1], [1, 2]]);;
<digraph with 2 vertices, 3 edges>
gap> ran := Digraph([[1], [1, 2], [1, 3]]);
<digraph with 3 vertices, 5 edges>
gap> IsDigraphMonomorphism(src, ran, ());
true
gap> IsDigraphEmbedding(src, ran, ());
true
gap> ran := Digraph([[1, 2], [1, 2], [1, 3]]);
<digraph with 3 vertices, 6 edges>
gap> IsDigraphMonomorphism(src, ran, IdentityTransformation);
true
gap> IsDigraphEmbedding(src, ran, IdentityTransformation);
false
```

### 7.3.12 GeneratorsOfEndomorphismMonoid

**GeneratorsOfEndomorphismMonoid**

*GeneratorsOfEndomorphismMonoid*(digraph[, colors][, limit])

*GeneratorsOfEndomorphismMonoidAttr*(digraph)

<table>
<thead>
<tr>
<th>Returns:</th>
<th>A list of transformations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>An endomorphism of digraph is a homomorphism \text{DigraphHomomorphism} (7.3.2) from digraph back to itself. \text{GeneratorsOfEndomorphismMonoid}, called with a single argument, returns a generating set for the monoid of all endomorphisms of digraph.</td>
<td></td>
</tr>
<tr>
<td>If the colors argument is specified, then \text{GeneratorsOfEndomorphismMonoid} 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:</td>
<td></td>
</tr>
<tr>
<td>• A list of positive integers of size the number of vertices of digraph, where colors[i] is the colour of vertex i.</td>
<td></td>
</tr>
<tr>
<td>• A list of lists, such that colors[i] is a list of all vertices with colour i.</td>
<td></td>
</tr>
<tr>
<td>If the limit argument is specified, then it will return only the first limit homomorphisms, where limit must be a positive integer or infinity.</td>
<td></td>
</tr>
</tbody>
</table>

Example

```gap
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);
```
7.3.13 DigraphColouring (for a digraph and a number of colours)

- DigraphColouring(digraph, n)
  - Returns: A transformation, or fail.

  A proper colouring of a digraph is a labelling of its vertices in such a way that adjacent vertices have different labels. A proper n-colouring is a proper colouring that uses exactly n colours. Equivalently, a proper (n-)colouring of a digraph can be defined to be a DigraphEpimorphism (7.3.6) from a digraph onto the complete digraph (with n vertices); see CompleteDigraph (3.5.2). Note that a digraph with loops (DigraphHasLoops (6.1.1)) does not have a proper n-colouring for any value n.

  If digraph is a digraph and n is a non-negative integer, then DigraphColouring(digraph, n) returns an epimorphism from digraph onto the complete digraph with n vertices if one exists, else it returns fail.

  See also DigraphGreedyColouring (7.3.14) and Note that a digraph with at least two vertices has a 2-colouring if and only if it is bipartite, see IsBipartiteDigraph (6.1.3).

Example

```gap
gap> DigraphColouring(CompleteDigraph(5), 4);
fail
gap> DigraphColouring(ChainDigraph(10), 1);
fail
gap> gr := ChainDigraph(10);;
Transformation([ 1, 2, 1, 2, 1, 2, 1, 2, 1, 2 ]);  
gap> t := DigraphColouring(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> DigraphGreedyColouring(gr);
Transformation([ 2, 1, 2, 1, 2, 1, 2, 1, 2, 1 ]);  
```

7.3.14 DigraphGreedyColouring (for a digraph and vertex order)

- DigraphGreedyColouring(digraph, order)
- DigraphGreedyColouring(digraph, func)
- DigraphGreedyColouring(digraph)

  Returns: A transformation, or fail.

  A proper colouring of a digraph is a labelling of its vertices in such a way that adjacent vertices have different labels. Note that a digraph with loops (DigraphHasLoops (6.1.1)) does not have any proper colouring.

  If digraph is a digraph and order is a dense list consisting of all of the vertices of digraph (in any order), then DigraphGreedyColouring uses a greedy algorithm with the specified order to obtain some proper colouring of digraph, which may not use the minimal number of colours.
If `digraph` is a digraph and `func` is a function whose argument is a digraph, and that returns a dense list `order`, then `DigraphGreedyColouring(digraph, func)` returns `DigraphGreedyColouring(digraph, func(digraph))`.

If the optional second argument (either a list or a function), is not specified, then `DigraphWelshPowellOrder (7.3.15)` is used by default.

See also `DigraphColouring (7.3.13)`.

```
Example

gap> DigraphGreedyColouring(ChainDigraph(10));
Transformation( [ 2, 1, 2, 1, 2, 1, 2, 1, 2, 1 ] )
gap> DigraphGreedyColouring(ChainDigraph(10), [1 .. 10]);
Transformation( [ 1, 2, 1, 2, 1, 2, 1, 2, 1, 2 ] )
```

### 7.3.15 DigraphWelshPowellOrder

> **DigraphWelshPowellOrder(digraph)** (attribute)

**Returns:** A list of the vertices.

`DigraphWelshPowellOrder` returns a list of all of the vertices of the digraph `digraph` ordered according to the sum of the number of out- and in-neighbours, from highest to lowest.

```
Example

 gap> DigraphWelshPowellOrder(Digraph([[4], [9], [9], []],[4, 6, 9], [1], [], [4, 5], [4, 5]]));
[ 5, 9, 4, 1, 6, 10, 2, 3, 7, 8 ]
```

### 7.3.16 ChromaticNumber

> **ChromaticNumber(digraph)** (attribute)

**Returns:** A non-negative integer.

A proper colouring of a digraph is a labelling of its vertices in such a way that adjacent vertices have different labels. Equivalently, a proper digraph colouring can be defined to be a DigraphEpimorphism (7.3.6) from a digraph onto a complete digraph.

If `digraph` is a digraph without loops (see `DigraphHasLoops (6.1.1)`), then `ChromaticNumber` returns the least non-negative integer `n` such that there is a proper colouring of `digraph` with `n` colours. In other words, for a digraph with at least one vertex, `ChromaticNumber` returns the least number `n` such that `DigraphColouring(digraph, n)` does not return `fail`. See `DigraphColouring (7.3.13)`.

```
Example

 gap> ChromaticNumber(NullDigraph(10));
1
gap> ChromaticNumber(CompleteDigraph(10));
10
gap> ChromaticNumber(CompleteBipartiteDigraph(5, 5));
2
gap> ChromaticNumber(Digraph([[], [3], [5], [2, 3], [4]]));
3
gap> ChromaticNumber(NullDigraph(0));
0
```
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 (6.1.8) for more information about these properties. Therefore given a digraph digraph, the cliques and independent sets of digraph are equal to the cliques and independent sets of the digraph:

- DigraphRemoveLoops(DigraphRemoveAllMultipleEdges(digraph)).

See DigraphRemoveLoops (3.3.23) and DigraphRemoveAllMultipleEdges (3.3.24) 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 (7.2.2) 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 (3.3.10). 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 
(3.3.9), 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 independent 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
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

\texttt{\textbackslash{}\textbackslash{} CliquesFinder(digraph, hook, user\_param, limit, include, exclude, max, size, reps)} (function)

\textbf{Returns:} The argument \texttt{user\_param}.

This function finds cliques of the digraph \texttt{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 the maximal symmetric subdigraph of \texttt{digraph} without loops (see AutomorphismGroup (7.2.2) and MaximalSymmetricSubdigraphWithoutLoops (3.3.4)).

\texttt{hook}

This argument should be a function or \texttt{fail}.

If \texttt{hook} is a function, then it should have two arguments \texttt{user\_param} (see below) and a clique \( c \). The function \texttt{hook(user\_param, c)} is called every time a new clique \( c \) is found by CliquesFinder.

If \texttt{hook} is \texttt{fail}, then a default function is used which simply adds every new clique found by CliquesFinder to \texttt{user\_param}, which must be a list in this case.

\texttt{user\_param}

If \texttt{hook} is a function, then \texttt{user\_param} can be any GAP object. The object \texttt{user\_param} is used as the first argument for the function \texttt{hook}. For example, \texttt{user\_param} might be a list, and \texttt{hook(user\_param, c)} might add the size of the clique \( c \) to the list \texttt{user\_param}.

If the value of \texttt{hook} is \texttt{fail}, then the value of \texttt{user\_param} must be a list.

\texttt{limit}

This argument should be a positive integer or \texttt{infinity}. CliquesFinder will return after it has found \texttt{limit} cliques or the search is complete.

\texttt{include} and \texttt{exclude}

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

CliquesFinder will only look for cliques containing all of the vertices in \texttt{include} and containing none of the vertices in \texttt{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.

\texttt{max}

This argument should be \texttt{true} or \texttt{false}. If \texttt{max} is \texttt{true} then CliquesFinder will only search for \texttt{maximal} cliques. If \texttt{max} is \texttt{false} then non-maximal cliques may be found.

\texttt{size}

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

\texttt{reps}

This argument should be \texttt{true} or \texttt{false}. 

If \( \text{reps} \) is true then the arguments \textit{include} and \textit{exclude} are each required to be invariant under the action of \( G \) on sets of vertices. In this case, \texttt{Cliqu.esFinder} will find representatives of the orbits of the desired cliques under the action of \( G \), \textit{although representatives may be returned which are in the same orbit}. If \( \text{reps} \) is false then \texttt{Cliqu.esFinder} will not take this into consideration.

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

\begin{verbatim}
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;;
gap> Cliqu.esFinder(gr, f, user_param, infinity, [], [], false, fail, true);
[ 1, 2, 3, 4, 5 ]
gap> Cliqu.esFinder(gr, fail, [], 5, [2, 4], [3], false, fail, fail);
[ [ 2, 4 ], [ 1, 2, 4 ], [ 2, 4, 5 ], [ 1, 2, 4, 5 ] ]
gap> Cliqu.esFinder(gr, fail, [], 2, [2, 4], [3], false, fail, fail);
[ [ 2, 4 ], [ 1, 2, 4 ], [ 2, 4, 5 ] ]
gap> Cliqu.esFinder(gr, fail, [], infinity, [], [], true, 5, false);
[ [ 1, 2, 3, 4, 5 ] ]
gap> Cliqu.esFinder(gr, fail, [], infinity, [1, 3], [], false, 3, false);
[ [ 1, 2, 3 ], [ 1, 3, 4 ], [ 1, 3, 5 ] ]
gap> Cliqu.esFinder(gr, fail, [], infinity, [1, 3], [], true, 3, false);
[ ]
\end{verbatim}
```

### 8.1.3 DigraphClique

\begin{verbatim}
D i g r a p h C l i q u e ( digraph[, include[, exclude[, size]]])
\end{verbatim}

**Returns:** A list of positive integers, or \texttt{fail}.

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

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

If \textit{include} is not a clique, then these functions return \texttt{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 \texttt{DigraphClique} and \texttt{DigraphMaximalClique} greedily enlarge the clique \textit{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 DigraphMaximalClique (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 DigraphMaximalClique(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 DigraphMaximalClique(digraph, include, exclude, 1, size) if one exists, else fail.

**Example**

```gap
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 DigraphMaximalClique**

- DigraphMaximalClique(digraph[, include[, exclude[, limit[, size]]]]) (function)
- DigraphMaximalCliqueReps(digraph[, include[, exclude[, limit[, size]]]]) (function)
- DigraphCliques(digraph[, include[, exclude[, limit[, size]]]]) (function)
- DigraphCliquesReps(digraph[, include[, exclude[, limit[, size]]]]) (function)
- DigraphMaximalCliqueAttr(digraph) (attribute)
- DigraphMaximalCliqueRepsAttr(digraph) (attribute)

**Returns:** A list of lists of positive integers.
If \texttt{digraph} is digraph, then these functions and attributes use \texttt{CliqueFinder (8.1.2)} to return cliques of \texttt{digraph}. A clique is defined by the set of vertices which it contains; see \texttt{IsClique (8.1.1)} and \texttt{IsMaximalClique (8.1.1)}.

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

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

Let \texttt{G} denote the automorphism group of maximal symmetric subdigraph of \texttt{digraph} without loops (see \texttt{AutomorphismGroup (7.2.2)} and \texttt{MaximalSymmetricSubdigraphWithoutLoops (3.3.4)}).

**Distinct cliques**

\texttt{DigraphMaximalClique} and \texttt{DigraphClique} each return a duplicate-free list of at most \texttt{limit} cliques of \texttt{digraph} which satisfy the arguments.

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

**Orbit representatives of cliques**

To use \texttt{DigraphMaximalCliqueReps} or \texttt{DigraphCliqueReps}, the arguments \texttt{include} and \texttt{exclude} must each be invariant under the action of \texttt{G} on sets of vertices.

If this is the case, then \texttt{DigraphMaximalCliqueReps} and \texttt{DigraphCliqueReps} each return a duplicate-free list of at most \texttt{limit} orbits representatives (under the action of \texttt{G} on sets vertices) of cliques of \texttt{digraph} which satisfy the arguments.

The representatives are not guaranteed to be in distinct orbits. However, if \texttt{lim} is not specified, or fewer than \texttt{lim} results are returned, then there will be at least one representative from each orbit of maximal cliques.

```gap
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> DigraphMaximalClique(gr);
[ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 3, 4 ] ]
gap> DigraphMaximalCliqueReps(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> DigraphMaximalCliqueReps(gr, [3, 4], [], 1);
```

---

Example
Digraphs

8.1.5 CliqueNumber

\[ \text{CliqueNumber}(\text{digraph}) \] (attribute)

Returns: A non-negative integer.

If \text{digraph} is a digraph, then \text{CliqueNumber}(\text{digraph}) returns the largest integer \( n \) such that \text{digraph} contains a clique with \( n \) vertices as an induced subdigraph.

A clique of a digraph is a set of mutually adjacent vertices of the digraph. Loops and multiple edges are ignored for the purpose of determining the clique number of a digraph.

Example

\[
\begin{align*}
gap> \text{gr} &:= \text{CompleteDigraph}(4);; \\
gap> \text{CliqueNumber}(\text{gr}); \\
4 \\
gap> \text{gr} &:= \text{Digraph}([\{1, 2, 4, 4\}, \{1, 3, 4\}, \{2, 1\}, \{1, 2\}]); \\
<\text{multidigraph with 4 vertices, 11 edges}> \\
gap> \text{CliqueNumber}(\text{gr}); \\
3
\end{align*}
\]

8.2 Finding independent sets

8.2.1 IsIndependentSet

\[ \text{IsIndependentSet}(\text{digraph}, l) \] (operation)

\[ \text{IsMaximalIndependentSet}(\text{digraph}, l) \] (operation)

Returns: true or false.

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

\[
\begin{align*}
gap> \text{gr} &:= \text{CycleDigraph}(4);; \\
gap> \text{IsIndependentSet}(\text{gr}, [1]); \\
true \\
gap> \text{IsMaximalIndependentSet}(\text{gr}, [1]); \\
false \\
gap> \text{IsIndependentSet}(\text{gr}, [1, 4, 3]); \\
false \\
gap> \text{IsIndependentSet}(\text{gr}, [2, 4]); \\
true \\
gap> \text{IsMaximalIndependentSet}(\text{gr}, [2, 4]); \\
true
\end{align*}
\]
8.2.2 DigraphIndependentSet

- DigraphIndependentSet(digraph[, include[, exclude[, size]]])
- DigraphMaximalIndependentSet(digraph[, include[, exclude[, size]]])

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

If `digraph` is a digraph, then these functions returns an 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 an 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 an 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.

```gap
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 ]
```
8.2.3 DigraphMaximalIndependentSets

\[ \text{DigraphMaximalIndependentSets}(\text{digraph}[, \text{include}[\text{, exclude}[\text{, limit}[\text{, size}]]]]) \]

\[ \text{DigraphMaximalIndependentSetsReps}(\text{digraph}[, \text{include}[\text{, exclude}[\text{, limit}[\text{, size}]]]]) \]

\[ \text{DigraphIndependentSets}(\text{digraph}[, \text{include}[\text{, exclude}[\text{, limit}[\text{, size}]]]]) \]

\[ \text{DigraphIndependentSetsReps}(\text{digraph}[, \text{include}[\text{, exclude}[\text{, limit}[\text{, size}]]]]) \]

\[ \text{DigraphMaximalIndependentSetsAttr}(\text{digraph}) \]

\[ \text{DigraphMaximalIndependentSetsRepsAttr}(\text{digraph}) \]

**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 (7.2.2) of the DigraphSymmetricClosure (3.3.10) 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 list of at most \( \text{limit} \) orbits representatives (under the action of \( G \) on sets of vertices) of independent sets of digraph which satisfy the arguments.

The representatives are not guaranteed to be in distinct orbits. However, if \( \text{lim} \) is not specified, or fewer than \( \text{lim} \) results are returned, then there will be at least one representative from each orbit of maximal independent sets.

Example

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

gap> DigraphIndependentSetsReps(gr);
[ [ 1 ], [ 1, 3 ] ]

gap> Set(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 LaT\TeX text file and you must 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 LaT\TeX 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 opts.type.
**engine**

This option can be used to specify the GraphViz engine to use to render a .dot document. The valid choices are "dot", "neato", "circo", "twopi", "fdp", "sfdp", and "patchwork". Please refer to the GraphViz documentation for details on these engines. The default value for this option is "dot", and it must be specified in `opt.engine`.

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

### 9.1.2 DotDigraph

\[ \text{\texttt{DotDigraph}(digraph)} \]

\[ \text{\texttt{DotVertexLabelledDigraph}(digraph)} \]

**Returns:** A string.

DotDigraph 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.

DotVertexLabelledDigraph differs from DotDigraph only in that the values in `DigraphVertexLabels` (5.1.9) are used to label the vertices in the produced picture rather than the numbers 1 to the number of vertices of the digraph.

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](http://www.graphviz.org).

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

Example

```gap
gap> adj := List([1 .. 4], x -> [1, 1, 1, 1]);  
[ [ 1, 1, 1, 1 ], [ 1, 1, 1, 1 ], [ 1, 1, 1, 1 ], [ 1, 1, 1, 1 ] ]

gap> adj[1][3] := 0;
0

gap> gr := DigraphByAdjacencyMatrix(adj);
<digraph with 4 vertices, 15 edges>

gap> FileString("dot/k4.dot", DotDigraph(gr));
154
```

### 9.1.3 DotSymmetricDigraph

\[ \text{\texttt{DotSymmetricDigraph}(digraph)} \]

**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` (6.1.10).
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.

Example

\begin{verbatim}
gap> star := Digraph([[2, 2, 3, 4], [1, 1], [1], [1, 4]]);
<multidigraph with 4 vertices, 9 edges>
gap> IsSymmetricDigraph(star);
true
gap> FileString("dot/star.dot", DotSymmetricDigraph(gr));
83
\end{verbatim}

9.1.4 DotPartialOrderDigraph

\begin{verbatim}
DotPartialOrderDigraph(digraph)
\end{verbatim}

Returns: A string.

This function produces a graphical representation of a partial order digraph digraph. DotPartialOrderDigraph will return an error if digraph is not a partial order digraph. See IsPartialOrderDigraph.

Since digraph is a partial order, it is both reflexive and transitive. The output of DotPartialOrderDigraph is the DotDigraph (9.1.2) of the DigraphReflexiveTransitiveReduction (3.3.12) of digraph.

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 DotPartialOrderDigraph can be written to a file using the command FileString.

Example

\begin{verbatim}
gap> poset := Digraph([[1, 4], [2], [2, 3, 4], [4]]);
\end{verbatim}

9.1.5 DotPreorderDigraph

\begin{verbatim}
DotPreorderDigraph(digraph)
\end{verbatim}

Returns: A string.

This function produces a graphical representation of a preorder digraph digraph. DotPreorderDigraph will return an error if digraph is not a preorder digraph. See IsPreorderDigraph.

A preorder digraph is reflexive and transitive but in general it is not anti-symmetric and may have strongly connected components containing more than one vertex. The QuotientDigraph (3.3.7) Q
obtained by forming the quotient of digraph by the partition of its vertices into the strongly connected components satisfies IsPartialOrderDigraph (6.1.14). Thus every vertex of $Q$ corresponds to a strongly connected component of digraph. The output of DotPreorderDigraph displays the DigraphReflexiveTransitiveReduction (3.3.12) of $Q$ with vertices displayed as rounded rectangles labelled by all of the vertices of digraph in the corresponding strongly connected component.

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 DotPreorderDigraph can be written to a file using the command FileString (GAPDoc: FileString).

Example

| gap> preset := Digraph([[1, 2, 4, 5], [1, 2, 4, 5], [3, 4], [4], [1, 2, 4, 5]]); |
| true |
| gap> FileString("dot/preset.dot", DotPreorderDigraph(gr)); |
| 83 |

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 characters. 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 increasing and decreasing edges, depending whether the edge has its source bigger than the range. Then both sets of edges are written separately in Sparse6 format with a separation symbol in between.

9.2.1 DigraphFromGraph6String

- DigraphFromGraph6String(str) (operation)
- DigraphFromDigraph6String(str) (operation)
Digraphs

- 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 corresponding 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.

```
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> DigraphFromGraph6String("&?");
<digraph with 0 vertices, 0 edges>
gap> DigraphFromGraph6String("&CQFG");
<digraph with 4 vertices, 6 edges>
gap> DigraphFromGraph6String("&IM[SrKLC~1hesbUF_"");
<digraph with 10 vertices, 51 edges>
```

- 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 (9.2.5).

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

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 (9.2.5).

```
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 DigraphFile

- DigraphFile(filename[, coder][, mode])
  (function)

**Returns:** An IO file object.

If `filename` is a string representing the name of a file, then `DigraphFile` returns an IO package file object for that file.
If the optional argument \texttt{coder} is specified and is a function which either encodes a digraph as a string, or decodes a string into a digraph, then this function will be used when reading or writing to the returned file object. If the optional argument \texttt{coder} is not specified, then the encoding of the digraphs in the returned file object must be specified in the file extension. The file extension must be one of: .\texttt{g6}, .\texttt{s6}, .\texttt{d6}, .\texttt{ds6}, .\texttt{txt}, .\texttt{p}, or .\texttt{pickle}; more details of these file formats is given below.

If the optional argument \texttt{mode} is specified, then it must be one of: "\texttt{w}" (for write), "\texttt{a}" (for append), or "\texttt{r}" (for read). If \texttt{mode} is not specified, then "\texttt{r}" is used by default.

If \texttt{filename} ends in one of: .\texttt{gz}, .\texttt{bz2}, or .\texttt{xz}, then the digraphs which are read from, or written to, the returned file object are decompressed, or compressed, appropriately.

The file object returned by \texttt{DigraphFile} can be given as the first argument for either of the functions \texttt{ReadDigraphs} (9.2.4) or \texttt{WriteDigraphs} (9.2.5). The purpose of this is to reduce the overhead of recreating the file object inside the functions \texttt{ReadDigraphs} (9.2.4) or \texttt{WriteDigraphs} (9.2.5) when, for example, reading or writing many digraphs in a loop.

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

\textbf{graph6 (.\texttt{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.

\textbf{sparse6 (.\texttt{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.

\textbf{digraph6 (.\texttt{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.

\textbf{disparse6 (.\texttt{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.

\textbf{plain text (.\texttt{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 \texttt{ReadPlainTextDigraph} (9.2.12) for a more flexible way to store digraphs in a plain text file.

\textbf{pickled (\texttt{.p} or \texttt{.pickle})}

Digraphs are pickled using the \texttt{IO} package. This is particularly good when the \texttt{DigraphGroup} (7.2.9) is non-trivial.

\begin{verbatim}
Example
\end{verbatim}

\begin{verbatim}
gap> filename := Concatenation(DIGRAPHS_Dir(), "/tst/out/man.d6.gz");;
gap> file := DigraphFile(filename, "w");;
\end{verbatim}
Digraphs

```
gap> for i in [1 .. 10] do
  > WriteDigraphs(file, Digraph([[1, 3], [2], [1, 2]]));
  > od;
gap> IO_Close(file);;
gap> file := DigraphFile(filename, "r");;
gap> ReadDigraphs(file, 9);
<digraph with 3 vertices, 5 edges>
```

9.2.4 ReadDigraphs

```
ReadDigraphs(filename[, decoder][, n])

Returns: A digraph, or a list of digraphs.

If filename is a string containing the name of a file containing encoded digraphs or an IO file object created using DigraphFile (9.2.3), then ReadDigraphs returns the digraphs encoded in the file as a list. Note that if filename is a compressed file, which has been compressed appropriately to give a filename extension of .gz, .bz2, or .xz, then ReadDigraphs can read filename without it first needing to be decompressed.

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 n is specified, then ReadDigraphs returns the n-th digraph encoded in the file filename.

If the optional argument decoder is not specified, then ReadDigraphs will deduce which decoder to use based on the filename extension of filename (after removing the compression-related filename extensions .gz, .bz2, and .xz). For example, if the filename extension is .g6, then ReadDigraphs will use the graph6 decoder DigraphFromGraph6String (9.2.1).

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
Digraphs

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 (9.2.12) 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 (7.2.9) is non-trivial.

Example

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

9.2.5 WriteDigraphs

WriteDigraphs(filename, digraphs[, encoder][, mode])

If digraphs is a list of digraphs or a digraph and filename is a string or an IO file object created using DigraphFile (9.2.3), then WriteDigraphs writes the digraphs to the file represented by filename. If the supplied filename ends in one of the extensions .gz, .bz2, or .xz, then 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 size.

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 (9.2.12) 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 (7.2.9) 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");
IO_OK
gap> ReadDigraphs(filename);
[ <digraph with 0 vertices, 0 edges>,
  <digraph with 3 vertices, 5 edges>,
  <digraph with 10 vertices, 51 edges> ]
```
9.2.6 IteratorFromDigraphFile

```plaintext
IteratorFromDigraphFile(filename[, decoder])
```

**Returns:** An iterator.

If `filename` is a string representing the name of a file containing encoded digraphs, then `IteratorFromDigraphFile` returns an iterator for which the value of `NextIterator` (Reference: `NextIterator`) is the next digraph encoded in the file.

If the optional argument `decoder` is specified and is a function which decodes a string into a digraph, then `IteratorFromDigraphFile` will use `decoder` to decode the digraphs contained in `filename`.

The purpose of this function is to easily allow looping over digraphs encoded in a file when loading all of the encoded digraphs would require too much memory.

To see what file types are available, see `WriteDigraphs` (9.2.5).

```plaintext
gap> filename := Concatenation(DIGRAPHS_Dir(), "/tst/out/man.d6.gz");;
gap> file := DigraphFile(filename, "w");;
gap> for i in [1 .. 10] do
  > WriteDigraphs(file, Digraph([[1, 3], [2], [1, 2]]));
  > od;
gap> IO_Close(file);
<iterator>
gap> for x in iter do od;
```

9.2.7 DigraphPlainTextLineEncoder

```plaintext
DigraphPlainTextLineEncoder(delimiter1[, delimiter2], offset)
DigraphPlainTextLineDecoder(delimiter1[, delimiter2], offset)
```

**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.

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

> TournamentLineDecoder(str)

Returns: A digraph.

This function takes a string str, decodes it, and then returns the tournament [see IsTournament (6.1.11)] 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

\[
gap> gr := TournamentLineDecoder("100001");
gap> dec(last);
gap> OutNeighbours(gr);
\]

9.2.9 AdjacencyMatrixUpperTriangleLineDecoder

> AdjacencyMatrixUpperTriangleLineDecoder(str)

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 ≤ i. In order words, the digraph will be topologically sorted.

Example

\[
gap> gr := AdjacencyMatrixUpperTriangleLineDecoder("100001");
gap> OutNeighbours(gr);
gap> gr := AdjacencyMatrixUpperTriangleLineDecoder("11111x111");
\]
Digraphs

9.2.10 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 ], [ ], [ ], [ ] ]
```

9.2.11 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>
```
9.2.12 WritePlainTextDigraph

\begin{verbatim}
WritePlainTextDigraph(filename, digraph, delimiter, offset)
\end{verbatim}

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

\begin{verbatim}
gap> gr := Digraph([[1, 2, 3], [1, 1], [2]]);
gap> filename := Concatenation(DIGRAPHS_Dir(), "/tst/out/plain.txt");
gap> WritePlainTextDigraph(filename, gr, ",", -1);
gap> ReadPlainTextDigraph(filename, ",", 1, ['/', '%']);
\end{verbatim}

9.2.13 WriteDIMACSDigraph

\begin{verbatim}
WriteDIMACSDigraph(filename, digraph)
\end{verbatim}

These operations write or read the single symmetric digraph `digraph` to or from a file in DIMACS format, as appropriate. The operation `WriteDIMACSDigraph` records the vertices and edges of `digraph`. The vertex labels of `digraph` will be recorded only if they are integers. See `IsSymmetricDigraph` (6.1.10) and `DigraphVertexLabels` (5.1.9).

The first argument `filename` should be the name of the file which will be written to or read from. A file can contain one symmetric digraph in DIMACS format. If `filename` ends in one of `.gz`, `.bz2`, or `.xz`, then the file is compressed, or decompressed, appropriately.

The DIMACS format is described as follows. Each line in the DIMACS file has one of four types:

- A line beginning with `c` and followed by any number of characters is a comment line, and is ignored.
- A line beginning with `p` defines the numbers of vertices and edges the digraph. This line has the format `p edge <nr_vertices> <nr_edges>`, where `<nr_vertices>` and `<nr_edges>` are replaced by the relevant integers. There must be exactly one such line in the file, and it must occur before any of the following kinds of line.
- A line of the form `e <v> <w>`, where `<v>` and `<w>` are integers in the range `[1 .. <nr_vertices>]`, specifies that there is a (symmetric) edge in the digraph between the ver-
tices $v$ and $w$. A symmetric edge only needs to be defined once; an additional line $e \ v \ w$, or $e \ w \ v$, will be interpreted as an additional, multiple, edge. Loops are permitted.

- A line of the form $n \ v \ label$, where $v$ is an integer in the range $[1 .. nr \_vertices]$ and $label$ is an integer, signifies that the vertex $v$ has the label $label$ in the digraph. If a label is not specified for a vertex, then ReadDIMACSDigraph will assign the label 1, according to the DIMACS specification.

A detailed definition of the DIMACS format can be found at http://mat.gsia.cmu.edu/COLOR/general/ccformat.ps, in Section 2.1. Note that optional descriptor lines, as described in Section 2.1, will be ignored.

```
Example

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

filename := Concatenation(DIGRAPHS_Dir(), 
> "tst/out/dimacs.dimacs");
gap> WriteDIMACSDigraph(filename, gr);
gap> ReadDIMACSDigraph(filename);
<digraph with 4 vertices, 8 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.

<table>
<thead>
<tr>
<th>Grape command</th>
<th>Digraphs command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph</td>
<td>Digraph (3.1.5)</td>
</tr>
<tr>
<td>EdgeOrbitsGraph</td>
<td>EdgeOrbitsDigraph (3.1.8)</td>
</tr>
<tr>
<td>NullGraph</td>
<td>NullDigraph (3.5.6)</td>
</tr>
<tr>
<td>CompleteGraph</td>
<td>CompleteDigraph (3.5.2)</td>
</tr>
<tr>
<td>JohnsonGraph</td>
<td>JohnsonDigraph (3.5.7)</td>
</tr>
<tr>
<td>CayleyGraph</td>
<td>CayleyDigraph (3.1.10)</td>
</tr>
<tr>
<td>AddEdgeOrbit</td>
<td>DigraphAddEdgeOrbit (3.3.16)</td>
</tr>
<tr>
<td>RemoveEdgeOrbit</td>
<td>DigraphRemoveEdgeOrbit (3.3.21)</td>
</tr>
<tr>
<td>AssignVertexNames</td>
<td>SetDigraphVertexLabels (5.1.9)</td>
</tr>
</tbody>
</table>

A.2 Functions to inspect graphs, vertices and edges

The table in this section contains more information when viewed in html format.
A.3 Functions to determine regularity properties of graphs

The table in this section contains more information when viewed in html format.

<table>
<thead>
<tr>
<th>Grape command</th>
<th>Digraphs command</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsRegularGraph</td>
<td>IsOutRegularDigraph (6.2.2)</td>
</tr>
<tr>
<td>LocalParameters</td>
<td>None</td>
</tr>
<tr>
<td>GlobalParameters</td>
<td>None</td>
</tr>
<tr>
<td>IsDistanceRegular</td>
<td>IsDistanceRegularDigraph (6.2.4)</td>
</tr>
<tr>
<td>CollapsedAdjacencyMat</td>
<td>None</td>
</tr>
<tr>
<td>OrbitalGraphColadjMats</td>
<td>None</td>
</tr>
<tr>
<td>VertexTransitiveDRGs</td>
<td>None</td>
</tr>
</tbody>
</table>

A.4 Some special vertex subsets of a graph

The table in this section contains more information when viewed in html format.

<table>
<thead>
<tr>
<th>Grape command</th>
<th>Digraphs command</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectedComponent</td>
<td>DigraphConnectedComponent (5.3.9)</td>
</tr>
<tr>
<td>ConnectedComponents</td>
<td>DigraphConnectedComponents (5.3.8)</td>
</tr>
<tr>
<td>Bicomponents</td>
<td>DigraphBicomponents (5.3.12)</td>
</tr>
<tr>
<td>DistanceSet</td>
<td>DigraphDistanceSet (5.3.5)</td>
</tr>
<tr>
<td>Layers</td>
<td>DigraphLayers (5.3.22)</td>
</tr>
<tr>
<td>IndependentSet</td>
<td>DigraphIndependentSet (8.2.2)</td>
</tr>
</tbody>
</table>
A.5 Functions to construct new graphs from old

The table in this section contains more information when viewed in html format.

<table>
<thead>
<tr>
<th>Grape command</th>
<th>Digraphs command</th>
</tr>
</thead>
<tbody>
<tr>
<td>InducedSubgraph</td>
<td>InducedSubdigraph (3.3.2)</td>
</tr>
<tr>
<td>DistanceSetInduced</td>
<td>None</td>
</tr>
<tr>
<td>DistanceGraph</td>
<td>DistanceDigraph (3.3.34)</td>
</tr>
<tr>
<td>ComplementGraph</td>
<td>DigraphDual (3.3.9)</td>
</tr>
<tr>
<td>PointGraph</td>
<td>None</td>
</tr>
<tr>
<td>EdgeGraph</td>
<td>EdgeUndirectedDigraph (3.3.30)</td>
</tr>
<tr>
<td>SwitchedGraph</td>
<td>None</td>
</tr>
<tr>
<td>UnderlyingGraph</td>
<td>DigraphSymmetricClosure (3.3.10)</td>
</tr>
<tr>
<td>QuotientGraph</td>
<td>QuotientDigraph (3.3.7)</td>
</tr>
<tr>
<td>BipartiteDouble</td>
<td>BipartiteDoubleDigraph (3.3.32)</td>
</tr>
<tr>
<td>GeodesicsGraph</td>
<td>None</td>
</tr>
<tr>
<td>CollapsedIndependentOrbitsGraph</td>
<td>None</td>
</tr>
<tr>
<td>CollapsedCompleteOrbitsGraph</td>
<td>None</td>
</tr>
<tr>
<td>NewGroupGraph</td>
<td>None</td>
</tr>
</tbody>
</table>

A.6 Vertex-Colouring and Complete Subgraphs

The table in this section contains more information when viewed in html format.

<table>
<thead>
<tr>
<th>Grape command</th>
<th>Digraphs command</th>
</tr>
</thead>
<tbody>
<tr>
<td>VertexColouring</td>
<td>DigraphGreedyColouring (7.3.14)</td>
</tr>
<tr>
<td>CompleteSubgraphs</td>
<td>DigraphCliques (8.1.4)</td>
</tr>
<tr>
<td>CompleteSubgraphsOfGivenSize</td>
<td>DigraphCliques (8.1.4)</td>
</tr>
</tbody>
</table>

A.7 Automorphism groups and isomorphism testing for graphs

The table in this section contains more information when viewed in html format.

<table>
<thead>
<tr>
<th>Grape command</th>
<th>Digraphs command</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutGroupGraph</td>
<td>AutomorphismGroup (7.2.2)</td>
</tr>
<tr>
<td>GraphIsomorphism</td>
<td>IsomorphismDigraphs (7.2.16)</td>
</tr>
<tr>
<td>IsIsomorphicGraph</td>
<td>IsIsomorphicDigraph (7.2.14)</td>
</tr>
<tr>
<td>GraphIsomorphismClass Representatives</td>
<td>None</td>
</tr>
</tbody>
</table>
References


Index

< (for digraphs), 37
= (for digraphs), 37
AdjacencyMatrix, 47
AdjacencyMatrixMutableCopy, 47
AdjacencyMatrixUpperTriangleLineDecoder, 136
ArticulationPoints, 60
AsBinaryRelation, 15
AsDigraph, 15
AsGraph, 16
AsMonoid, 68
AsSemigroup, 68
AsTransformation, 17
AutomorphismGroup
  for a digraph, 91
  for a digraph and a homogeneous list, 93
BipartiteDoubleDigraph, 32
BlissAutomorphismGroup, 92
BlissCanonicalDigraph, 97
BlissCanonicalLabeling
  for a digraph, 94
  for a digraph and a list, 95
BooleanAdjacencyMatrix, 48
BooleanAdjacencyMatrixMutableCopy, 48
CayleyDigraph, 14
ChainDigraph, 34
CharacteristicPolynomial, 47
ChromaticNumber, 115
CliqueNumber, 122
CliqueFinder, 118
CompleteBipartiteDigraph, 35
CompleteDigraph, 34
CompleteMultipartiteDigraph, 35
CycleDigraph, 35
Digraph, 11
  for a group, list, function, and function, 11
  for a list and function, 11
Digraph6String, 130
DigraphAddAllLoops, 32
DigraphAddEdge, 24
DigraphAddEdgeOrbit, 25
DigraphAddEdges, 25
DigraphAddVertex, 23
DigraphAddVertices, 24
DigraphAdjacencyFunction, 48
DigraphAllSimpleCircuits, 64
DigraphBiconnectedComponents, 59
DigraphByAdjacencyMatrix, 13
DigraphByEdges, 13
DigraphByInNeighbors, 14
DigraphByInNeighbours, 14
DigraphClique, 119
DigraphCliqueSubset, 120
DigraphCliqueSubsetReps, 120
DigraphClosure
  for a digraph and positive integer, 33
DigraphColoring
  for a digraph and a number of colours, 114
DigraphColouring
  for a digraph and a number of colours, 114
DigraphConnectedComponent, 58
DigraphConnectedComponents, 58
DigraphCopy, 17
DigraphDegeneracy, 65
DigraphDegeneracyOrdering, 66
DigraphDiameter, 54
DigraphDisjointUnion
  for a list of digraphs, 29
  for an arbitrary union of digraphs, 29
DigraphDistanceSet
  for a digraph, a pos int, and a list, 56
  for a digraph, a pos int, and an int, 56
DigraphDual, 21
DigraphEdgeLabel, 44
Digraphs

DigraphEdgeLabels, 44
DigraphEdges, 41
DigraphEdgeUnion
  for a list of digraphs, 30
  for an arbitrary number of digraphs, 30
DigraphEmbedding, 111
DigraphEpimorphism, 110
DigraphFamily, 10
DigraphFile, 130
DigraphFloydWarshall, 61
DigraphFromDigraph6String, 129
DigraphFromDiSparse6String, 130
DigraphFromGraph6String, 129
DigraphFromPlainTextString, 137
DigraphFromSparse6String, 130
DigraphGirth, 57
DigraphGreedyColouring
  for a digraph, 114
  for a digraph and vertex order, 114
  for a digraph and vertex order function, 114
DigraphGroup, 97
DigraphHasLoops, 73
DigraphHomomorphism, 108
DigraphIndependentSet, 123
DigraphIndependentSets, 124
DigraphIndependentSetsReps, 124
DigraphInEdges, 45
DigraphJoin
  for a list of digraphs, 30
  for an arbitrary number of digraphs, 30
DigraphLayers, 65
DigraphLongestDistanceFromVertex, 56
DigraphLongestSimpleCircuit, 65
DigraphLoops, 53
DigraphMaximalClique, 119
DigraphMaximalClique, 120
DigraphMaximalCliques, 120
DigraphMaximalCliquesAttr, 120
DigraphMaximalCliquesReps, 120
DigraphMaximalCliquesRepsAttr, 120
DigraphMaximalIndependentSet, 123
DigraphMaximalIndependentSets, 124
DigraphMaximalIndependentSetsAttr, 124
DigraphMaximalIndependentSetsReps, 124
DigraphMaximalIndependentSetsRepsAttr, 124
DigraphMonomorphism, 109
DigraphNrEdges, 41
DigraphNrVertices, 40
DigraphOrbitReps, 99
DigraphOrbits, 99
DigraphOutEdges, 45
DigraphPath, 62
DigraphPeriod, 60
DigraphPlainTextLineDecoder, 135
DigraphPlainTextLineEncoder, 135
DigraphRange, 49
DigraphReflexiveTransitiveClosure, 22
DigraphReflexiveTransitiveReduction, 23
DigraphRemoveAllMultipleEdges, 28
DigraphRemoveEdge, 26
DigraphRemoveEdgeOrbit, 27
DigraphRemoveEdges, 27
DigraphRemoveLoops, 28
DigraphRemoveVertex, 26
DigraphRemoveVertices, 26
DigraphReverse, 21
DigraphReverseEdge, 28
DigraphReverseEdges, 28
Digraphs package overview, 5
DigraphSchreierVector, 99
DigraphShortestDistance
  for a digraph and a list, 55
  for a digraph and two vertices, 55
  for a digraph, a list, and a list, 55
DigraphShortestDistances, 55
DigraphShortestPath, 63
DigraphSinks, 42
DigraphsMakeDoc, 9
DigraphSource, 49
DigraphSources, 42
DigraphStabilizer, 100
DigraphsTestInstall, 9
DigraphsTestStandard, 9
DigraphsUseBliss, 90
DigraphsUseNauty, 90
DigraphSymmetricClosure, 21
DigraphTopologicalSort, 42
DigraphTransitiveClosure, 22
DigraphTransitiveReduction, 23
DigraphType, 10
Digraphs

IsMeetSemilatticeDigraph, 79
IsMultiDigraph, 76
IsNullDigraph, 75
IsomorphismDigraphs
  for digraphs, 102
    for digraphs and homogeneous lists, 103
IsOuterPlanarDigraph, 88
IsOutRegularDigraph, 80
IsPartialOrderDigraph, 79
IsPerfectMatching, 46
IsPlanarDigraph, 87
IsPreorderDigraph, 78
IsQuasiorderDigraph, 78
IsReachable, 62
IsReflexiveDigraph, 76
IsRegularDigraph, 81
IsStronglyConnectedDigraph, 83
IsSubdigraph, 37
IsSymmetricDigraph, 77
IsTournament, 77
IsTransitiveDigraph, 78
IsUndirectedForest, 85
IsUndirectedSpanningForest, 38
IsUndirectedSpanningTree, 38
IsUndirectedTree, 85
IteratorFromDigraphFile, 135
IteratorOfPaths, 63
JohnsonDigraph, 36
KuratowskiOuterPlanarSubdigraph, 69
KuratowskiPlanarSubdigraph, 69
LineDigraph, 31
LineUndirectedDigraph, 31
MaximalAntiSymmetricSubdigraph, 19
MaximalSymmetricSubdigraph, 18
MaximalSymmetricSubdigraphWithoutLoops, 18
MonomorphismsDigraphs, 109
MonomorphismsDigraphsRepresentatives, 109
NautyAutomorphismGroup, 92
NautyCanonicalDigraph, 97
NautyCanonicalLabelling
  for a digraph, 94
  for a digraph and a list, 95
NullDigraph, 35
OnDigraphs
  for a digraph and a perm, 89
  for a digraph and a transformation, 89
OnMultiDigraphs, 90
  for a digraph, perm, and perm, 90
OutDegreeOfVertex, 52
OutDegrees, 51
OutDegreeSequence, 51
OutDegreeSet, 51
OuterPlanarEmbedding, 70
OutNeighbours, 49
OutNeighboursMutableCopy, 49
OutNeighboursOfVertex, 52
OutNeighbours, 49
OutNeighboursMutableCopy, 49
OutNeighboursOfVertex, 52
PartialOrderDigraphJoinOfVertices
  for a digraph and two vertices, 53
PartialOrderDigraphMeetOfVertices
  for a digraph and two vertices, 53
PlainTextString, 137
PlanarEmbedding, 70
QuotientDigraph, 20
RandomDigraph, 33
RandomMultiDigraph, 34
RandomTournament, 34
ReadDigraphs, 132
ReadDIMACSDigraph, 138
ReadPlainTextDigraph, 138
ReducedDigraph, 18
RepresentativeOutNeighbours, 104
SemigroupOfCayleyDigraph, 67
SetDigraphEdgeLabel, 44
SetDigraphEdgeLabels
  for a digraph and a function, 44
  for a digraph and a list of lists, 44
SetDigraphVertexLabel, 42
SetDigraphVertexLabels, 43
Sparse6String, 130
Splash, 126
SubdigraphHomeomorphicToK23, 71
SubdigraphHomeomorphicToK33, 71
SubdigraphHomeomorphicToK4, 71

TCodeDecoder, 137
TournamentLineDecoder, 136

UndirectedSpanningForest, 19
UndirectedSpanningTree, 19

WriteDigraphs, 133
WriteDIMACSDigraph, 138
WritePlainTextDigraph, 138