Digraphs
Version 1.1.1

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

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

Julius Jonušas  Email: julius.jonusas@tuwien.ac.at
Homepage: http://julius.jonusas.work

James D. Mitchell  Email: jdm3@st-and.ac.uk
Homepage: http://www-groups.mcs.st-andrews.ac.uk/~jamesm

Michael Torpey  Email: mct25@st-andrews.ac.uk
Homepage: https://mtorpey.github.io

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 ............................. 11
   3.1 Creating digraphs ............................. 11
   3.2 Changing representations .................... 17
   3.3 New digraphs from old ....................... 20
   3.4 Random digraphs ............................... 45
   3.5 Standard examples ............................. 47

4 Operators ............................................. 51
   4.1 Operators for digraphs ....................... 51

5 Attributes and operations ..................... 54
   5.1 Vertices and edges ............................ 54
   5.2 Neighbours and degree ....................... 62
   5.3 Reachability and connectivity ............... 71
   5.4 Cayley graphs of groups ..................... 87
   5.5 Associated semigroups .......................... 88
   5.6 Planarity ........................................ 89

6 Properties of digraphs ......................... 95
   6.1 Edge properties ............................... 95
   6.2 Regularity .................................... 104
   6.3 Connectivity and cycles ..................... 106
   6.4 Planarity ...................................... 112
   6.5 Homomorphisms and transformations .......... 114
7 Homomorphisms
7.1 Acting on digraphs .......................................................... 116
7.2 Isomorphisms and canonical labellings ................................. 117
7.3 Homomorphisms of digraphs ............................................. 133

8 Finding cliques and independent sets ................................. 145
8.1 Finding cliques .............................................................. 146
8.2 Finding independent sets .................................................. 152

9 Visualising and IO ............................................................. 156
9.1 Visualising a digraph ......................................................... 156
9.2 Reading and writing digraphs to a file ............................... 159

A Grape to Digraphs Command Map ..................................... 170
A.1 Functions to construct and modify graphs ......................... 170
A.2 Functions to inspect graphs, vertices and edges ................. 170
A.3 Functions to determine regularity properties of graphs ........ 171
A.4 Some special vertex subsets of a graph ............................ 171
A.5 Functions to construct new graphs from old ...................... 172
A.6 Vertex-Colouring and Complete Subgraphs ...................... 172
A.7 Automorphism groups and isomorphism testing for graphs .... 172

References ................................................................. 173

Index ................................................................. 174
Chapter 1

The Digraphs package

1.1 Introduction

This is the manual for version 1.1.1 of the Digraphs package. 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,
- Reinis Cirpons,
- Luke Elliott,
- Max Horn,
- Christopher Jefferson,
- Markus Pfeiffer,
- Chris Russell,
- Finn Smith, and
- Murray Whyte.

The Digraphs package contains a variety of methods for efficiently creating and storing mutable and immutable 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 (3.1.7). 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.

From version 1.0.0 of this package, digraphs can be either mutable or immutable. Mutable digraphs can be changed in-place by many of the methods in the package, which avoids unnecessary copying. Immutable digraphs cannot be changed in-place, but their advantage is that the value of an attribute of an immutable digraph is only ever computed once. Mutable digraphs can be converted into immutable digraphs in-place using MakeImmutable (Reference: MakeImmutable). One of the motivations for introducing mutable digraphs in version 1.0.0 was that in practice the authors often wanted to create a digraph and immediately modify it (removing certain edges, loops, and so on). Before version 1.0.0, this involved copying the digraph several times, with each copy being discarded almost immediately. From version 1.0.0, this unnecessary copying can be eliminated by first creating a mutable digraph, then changing it in-place, and finally converting the mutable digraph to an immutable one in-place (if desirable).

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-1.1.1.tar.gz from the Digraph package webpage.

• unzip and untar the file, this should create a directory called digraphs-1.1.1.

• locate the pkg directory of your GAP directory, which contains the directories lib, doc and so on. Move the directory digraphs-1.1.1 into the pkg directory.

• it is necessary to compile the Digraphs package. Inside the pkg/digraphs-1.1.1 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 (3.1.7) 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-1.1.1 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 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, please use the function `DigraphsMakeDoc` (2.4.1).

2.4.1 DigraphsMakeDoc

```
DigraphsMakeDoc()
```

*Return*: Nothing

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

2.5 Testing your installation

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

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

2.5.1 DigraphsTestInstall

```
DigraphsTestInstall()
```

*Return*: true or false.

This function can be called without arguments to test your installation of **Digraphs** is working correctly. These tests should take no more than a few seconds to complete. To test more comprehensively that **Digraphs** is working correctly, use `DigraphsTestStandard` (2.5.2).

2.5.2 DigraphsTestStandard

```
DigraphsTestStandard()
```

*Return*: true or false.

This function can be called without arguments to test all of the methods included in **Digraphs**. These tests should take less than a minute to complete.

To quickly test that **Digraphs** is installed correctly use `DigraphsTestInstall` (2.5.1). For a more thorough test, use `DigraphsTestExtreme` (2.5.3).

2.5.3 DigraphsTestExtreme

```
DigraphsTestExtreme()
```

*Return*: Nothing.

This function should be called with no argument. It executes a series of very demanding tests, which measure the performance of a variety of functions on large examples. These tests take a long time to complete, at least several minutes.
For these tests to complete, the digraphs library digraphs-lib must be downloaded and placed in the digraphs directory in a subfolder named digraphs-lib. This library can be found on the Digraphs website.
Chapter 3

Creating digraphs

In this chapter we describe how to create digraphs.

3.1 Creating digraphs

3.1.1 IsDigraph

▷ IsDigraph (Category)

Every digraph in Digraphs belongs to the category IsDigraph. Some basic attributes and operations for digraphs are DigraphVertices (5.1.1), DigraphEdges (5.1.3), and OutNeighbours (5.2.6).

3.1.2 IsMutableDigraph

▷ IsMutableDigraph (Category)

IsMutableDigraph is a synonym for IsDigraph (3.1.1) and IsMutable (Reference: IsMutable). A mutable digraph may be changed in-place by methods in the Digraphs package, and is not attribute-storing – see IsAttributeStoringRep (Reference: IsAttributeStoringRep).

A mutable digraph may be converted into an immutable attribute-storing digraph by calling MakeImmutable (Reference: MakeImmutable) on the digraph.

3.1.3 IsImmutableDigraph

▷ IsImmutableDigraph (Category)

IsImmutableDigraph is a subcategory of IsDigraph (3.1.1). Digraphs that lie in IsImmutableDigraph are immutable and attribute-storing. In particular, they lie in IsAttributeStoringRep (Reference: IsAttributeStoringRep).

A mutable digraph may be converted to an immutable digraph that lies in the category IsImmutableDigraph by calling MakeImmutable (Reference: MakeImmutable) on the digraph.

The operation DigraphMutableCopy (3.3.1) can be used to construct a mutable copy of an immutable digraph. It is however not possible to convert an immutable digraph into a mutable digraph in-place.
3.1.4 IsCayleyDigraph

IsCayleyDigraph is a subcategory of IsDigraph. Digraphs that are Cayley digraphs of a group and that are constructed by the operation CayleyDigraph (3.1.12) are constructed in this category, and are always immutable.

3.1.5 IsDigraphWithAdjacencyFunction

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

3.1.6 DigraphByOutNeighboursType

The type of all digraphs is DigraphByOutNeighboursType. The family of all digraphs is DigraphFamily.

3.1.7 Digraph

Returns: A digraph.

If the optional first argument filt is present, then this should specify the category or representation the digraph being created will belong to. For example, if filt is IsMutableDigraph (3.1.2), then the digraph being created will be mutable, if filt is IsImmutableDigraph (3.1.3), then the digraph will be immutable. If the optional first argument filt is not present, then IsImmutableDigraph (3.1.3) is used by default.

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 \( E^0 = [1 .. \text{Length}(obj)] \), 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 \([1 .. \text{Length}(obj)]\), then this function returns a digraph with vertices \( E^0 = [1 .. \text{Length}(obj)] \), 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 \( E^0 = [1 .. \text{obj}] \), 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).

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 \( \text{func(list[i], list[j])} \) returns true.

for a group, a list, and two functions
The arguments will be \( G, \text{list, act, adj} \).

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 (7.2.10), and is used to speed up operations like DigraphDiameter (5.3.1).

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

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

Example

```gap
> gr := Digraph([2, 5, 8, 10], [2, 3, 4, 2, 5, 6, 8, 9, 10], [1],
> [3, 5, 7, 8, 10], [2, 5, 7], [3, 6, 7, 9, 10], [1, 4],
> [1, 5, 9], [1, 2, 7, 8], [3, 5]);
<immutable multidigraph with 10 vertices, 38 edges>
> gr := Digraph(\["a", "b", "c"\], \["a"\], \["b"\]);
<immutable digraph with 3 vertices, 1 edge>
> gr := Digraph(5, [1, 2, 2, 4, 1, 1], [2, 3, 5, 5, 1, 1]);
<immutable multidigraph with 5 vertices, 6 edges>
> Petersen := Graph(SymmetricGroup(5), [[1, 2]], OnSets,
> function(x, y) return Intersection(x, y) = []; end);
> Digraph(Petersen);
```
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, \cite{vLS81}. The algebraic description is taken from the seminal paper of Calderbank and Kantor \cite{CK86}.

```
Example

gap> f := GF(3^4);
GF(3^4)

gap> gamma := First(f, x -> Order(x) = 5);
Z(3^4)^64

gap> L := Union([Zero(f)], List(Group(gamma)));
[ 0*Z(3), Z(3)^0, Z(3^4)^16, Z(3^4)^32, Z(3^4)^48, Z(3^4)^64 ]

gap> omega := Union(List(L, x -> List(Difference(L, [x]), y -> x - y)));
[ Z(3)^0, Z(3), Z(3^4)^5, Z(3^4)^7, Z(3^4)^8, Z(3^4)^13, Z(3^4)^15, Z(3^4)^16, Z(3^4)^21, Z(3^4)^23, Z(3^4)^24, Z(3^4)^29, Z(3^4)^31, Z(3^4)^32, Z(3^4)^37, Z(3^4)^39, Z(3^4)^45, Z(3^4)^47, Z(3^4)^48, Z(3^4)^53, Z(3^4)^55, Z(3^4)^56, Z(3^4)^61, Z(3^4)^63, Z(3^4)^64, Z(3^4)^69, Z(3^4)^71, Z(3^4)^72, Z(3^4)^77, Z(3^4)^79 ]

gap> adj := function(x, y)
> return x - y in omega;
> end;

function( x, y ) ... end

gap> digraph := Digraph(AsList(f), adj);
<immutable digraph with 81 vertices, 2430 edges>

gap> group := Group(Z(3));
<group with 1 generators>

gap> act := *;
<Operation "*">

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

### 3.1.8 DigraphByAdjacencyMatrix

\[ \Diamond \text{DigraphByAdjacencyMatrix}([\text{filt, } \text{list}]) \]  

\textbf{Returns:} A digraph.

If the optional first argument \textit{filt} is present, then this should specify the category or representation the digraph being created will belong to. For example, if \textit{filt} is IsMutableDigraph (\ref{IsMutableDigraph}), then the digraph being created will be mutable, if \textit{filt} is IsImmutableDigraph (\ref{IsImmutableDigraph}), then the digraph will be immutable. If the optional first argument \textit{filt} is not present, then IsImmutableDigraph (\ref{IsImmutableDigraph}) is used by default.

If \textit{list} is the adjacency matrix of a digraph in the sense of AdjacencyMatrix (\ref{AdjacencyMatrix}), then this operation returns the digraph which is defined by \textit{list}.

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

```
Example

gap> DigraphByAdjacencyMatrix([ 0, 1, 0, 2, 0],

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

\[
\begin{array}{c}
> [1, 1, 1, 0, 1], \\
> [0, 3, 2, 1, 1], \\
> [0, 0, 1, 0, 1], \\
> [2, 0, 0, 0, 0]); \\
<\text{immutable multidigraph with 5 vertices, 18 edges}>
\end{array}
\]

\[
\begin{array}{c}
gap> D := \text{DigraphByAdjacencyMatrix}([ \\
> [\text{true, false, true}], \\
> [\text{false, false, true}], \\
> [\text{false, true, false}]); \\
<\text{immutable digraph with 3 vertices, 4 edges}>
\end{array}
\]

\[
\begin{array}{c}
\text{OutNeighbours}(D); \\
[ [ 1, 3 ], [ 3 ], [ 2 ] ]
\end{array}
\]

\[
\begin{array}{c}
gap> D := \text{DigraphByAdjacencyMatrix}(<\text{IsMutableDigraph,} \\
> [[\text{true, false, true}], \\
> [\text{false, false, true}], \\
> [\text{false, true, false}]); \\
<\text{mutable digraph with 3 vertices, 4 edges}>
\end{array}
\]

3.1.9 DigraphByEdges

\[
<\text{DigraphByEdges}([\text{filt, } list[, n]]) (\text{operation})
\]

Returns: A digraph.

If the optional first argument \text{filt} is present, then this should specify the category or representation the digraph being created will belong to. For example, if \text{filt} is \text{IsMutableDigraph} (3.1.2), then the digraph being created will be mutable, if \text{filt} is \text{IsImmutableDigraph} (3.1.3), then the digraph will be immutable. If the optional first argument \text{filt} is not present, then \text{IsImmutableDigraph} (3.1.3) is used by default.

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

If the optional second argument \text{n} is a positive integer with \text{n} \geq \text{m} (with \text{m} defined as above), then this function returns the digraph with \text{n} vertices and list \text{list}.

See DigraphEdges (5.1.3).

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

\[
\begin{array}{c}
gap> \text{DigraphByEdges}(
> [[1, 3], [2, 1], [2, 3], [2, 5], [3, 6], \\
> [4, 6], [5, 2], [5, 4], [5, 6], [6, 6]], 12);
<\text{immutable digraph with 12 vertices, 10 edges}>
\end{array}
\]

\[
\begin{array}{c}
gap> \text{DigraphByEdges(IsMutableDigraph,} \\
> [[1, 3], [2, 1], [2, 3], [2, 5], [3, 6], \\
> [4, 6], [5, 2], [5, 4], [5, 6], [6, 6]], 12);
<\text{mutable digraph with 12 vertices, 10 edges}>
\end{array}
\]

3.1.10 EdgeOrbitsDigraph

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

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

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

Example

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

3.1.11 DigraphByInNeighbours

Example

```gap
D := DigraphByInNeighbours(2, 3, 2, 1, 2, 3);
<immutable multidigraph with 3 vertices, 7 edges>
```

3.1.12 CayleyDigraph

Example

```gap
D := CayleyDigraph(IsMutableDigraph, [2, 3, 2, 1, 2, 3]);
<mutable multidigraph with 3 vertices, 7 edges>
```
of $G$. There exists an edge from the vertex $u$ to the vertex $v$ if and only if there exists a generator $g$ in $gens$ such that $x \ast g = y$.

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

The digraph created by this operation belongs to the category IsCayleyDigraph (3.1.4), the group $G$ can be recovered from the digraph using GroupOfCayleyDigraph (5.4.1), and the generators $gens$ can be obtained using GeneratorsOfCayleyDigraph (5.4.2).

Note that this function can only return an immutable digraph.

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

### 3.2 Changing representations

#### 3.2.1 AsBinaryRelation

- **AsBinaryRelation** *(digraph)*

  **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*.

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

#### 3.2.2 AsDigraph

- **AsDigraph** *(filt, [trans[, n]])*

  **Returns:** A digraph, or fail.

  If the optional first argument *filt* is present, then this should specify the category or representation the digraph being created will belong to. For example, if *filt* is IsMutableDigraph (3.1.2), then the digraph being created will be mutable, if *filt* is IsImmutableDigraph (3.1.3), then the digraph will be immutable. If the optional first argument *filt* is not present, then IsImmutableDigraph (3.1.3) is used by default.
If \( \text{trans} \) is a transformation, and \( n \) is a non-negative integer such that the restriction of \( \text{trans} \) to \([1 \ldots n]\) defines a transformation of \([1 \ldots n]\), then \( \text{AsDigraph} \) returns the functional digraph with \( n \) vertices defined by \( \text{trans} \). See \( \text{IsFunctionalDigraph} \) (6.1.9).

Specifically, the digraph returned by \( \text{AsDigraph} \) has \( n \) edges: for each vertex \( x \) in \([1 \ldots n]\), there is a unique edge with source \( x \); this edge has range \( x^\text{trans} \).

If the optional second argument \( n \) is not supplied, then the degree of the transformation \( \text{trans} \) is used by default. If the restriction of \( \text{trans} \) to \([1 \ldots n]\) does not define a transformation of \([1 \ldots n]\), then \( \text{AsDigraph}(\text{trans}, n) \) returns \text{fail}.

Example

\[
\begin{verbatim}
gap> f := Transformation([4, 3, 3, 1, 7, 9, 10, 4, 2, 3]);
Transformation( [ 4, 3, 3, 1, 7, 9, 10, 4, 2, 3 ] )
gap> AsDigraph(f);
<immutable functional digraph with 10 vertices>
gap> AsDigraph(f, 4);
<immutable functional digraph with 4 vertices>
gap> AsDigraph(f, 5);
fail
\end{verbatim}
\]

3.2.3 Graph

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

Returns: A \textit{Grape} package graph.

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

If \( \text{digraph} \) is a multidigraph, then since \textit{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 \( \text{Graph}(\text{DigraphRemoveAllMultipleEdges}(\text{digraph})) \).

Example

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

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

**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 if \text{digraph} is immutable, then the result will be stored as a mutable attribute of \text{digraph}. In this latter case, when \text{AsGraph}( \text{digraph} ) is called subsequently, the same GAP object will be returned as before.

Example

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

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

**Returns:** A transformation, or \text{fail}

If \text{digraph} is a functional digraph, then \text{AsTransformation} returns the transformation which is defined by \text{digraph}. See \text{IsFunctionalDigraph} (6.1.9). Otherwise, \text{AsTransformation}( \text{digraph} ) returns \text{fail}.

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

Example

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

3.3 New digraphs from old

3.3.1 DigraphImmutableCopy

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DigraphImmutableCopy(digraph)</td>
<td>(operation)</td>
</tr>
<tr>
<td>DigraphMutableCopy(digraph)</td>
<td>(operation)</td>
</tr>
<tr>
<td>DigraphCopySameMutability(digraph)</td>
<td>(operation)</td>
</tr>
<tr>
<td>DigraphCopy(digraph)</td>
<td>(operation)</td>
</tr>
</tbody>
</table>

**Returns:** A digraph.

Each of these operations returns a new copy of `digraph`, of the appropriate mutability, retaining none of the attributes or properties of `digraph`.

DigraphCopy is a synonym for DigraphCopySameMutability.

**Example**

```gap
gap> D := CycleDigraph(10);  
<immutable cycle digraph with 10 vertices>  
gap> DigraphCopy(D) = D;  
true  
gap> IsIdenticalObj(DigraphCopy(D), D);  
false  
gap> DigraphMutableCopy(D);  
<mutable digraph with 10 vertices, 10 edges>
```

3.3.2 DigraphImmutableCopyIfImmutable

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DigraphImmutableCopyIfImmutable(digraph)</td>
<td>(operation)</td>
</tr>
<tr>
<td>DigraphImmutableCopyIfMutable(digraph)</td>
<td>(operation)</td>
</tr>
<tr>
<td>DigraphMutableCopyIfMutable(digraph)</td>
<td>(operation)</td>
</tr>
<tr>
<td>DigraphMutableCopyIfImmutable(digraph)</td>
<td>(operation)</td>
</tr>
</tbody>
</table>

**Returns:** A digraph.

Each of these operations returns either the original argument `digraph`, or a new copy of `digraph` of the appropriate mutability, according to the mutability of `digraph`.

**Example**

```gap
gap> C := CycleDigraph(10);  
<immutable cycle digraph with 10 vertices>  
gap> D := DigraphImmutableCopyIfImmutable(C);  
<immutable digraph with 10 vertices, 10 edges>  
gap> IsIdenticalObj(C, D);  
false  
gap> C = D;  
true  
gap> D := DigraphImmutableCopyIfMutable(C);  
<immutable cycle digraph with 10 vertices>  
gap> IsIdenticalObj(C, D);  
true  
gap> C = D;  
true  
gap> D := DigraphMutableCopyIfMutable(C);  
<immutable cycle digraph with 10 vertices>  
gap> IsMutableDigraph(D);  
false
```
Digraphs

3.3.3 InducedSubdigraph

\> \texttt{InducedSubdigraph(digraph, verts)}

(\textit{operation})

\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 \texttt{DigraphVertexLabels (5.1.9)}.

If \textit{digraph} belongs to \texttt{IsMutableDigraph (3.1.2)}, then \textit{digraph} is modified in place. If \textit{digraph} belongs to \texttt{IsImmutableDigraph (3.1.3)}, a new immutable digraph containing the appropriate vertices and edges is returned.

Example

\begin{verbatim}
gap> D := DigraphMutableCopyIfImmutable(C);
<mutable digraph with 10 vertices, 10 edges>
gap> IsMutableDigraph(D);
true
gap> C := CycleDigraph(IsMutableDigraph, 10);
<mutable digraph with 10 vertices, 10 edges>
gap> D := DigraphImmutableCopyIfImmutable(C);
<mutable digraph with 10 vertices, 10 edges>
gap> IsIdenticalObj(C, D);
true
gap> C = D;
true
gap> D := DigraphImmutableCopyIfMutable(C);
<mutable digraph with 10 vertices, 10 edges>
gap> IsIdenticalObj(C, D);
false
gap> C = D;
true
gap> D := DigraphMutableCopyIfMutable(C);
<mutable digraph with 10 vertices, 10 edges>
gap> IsMutableDigraph(D);
true
gap> D := DigraphMutableCopyIfImmutable(C);
<mutable digraph with 10 vertices, 10 edges>
gap> IsIdenticalObj(C, D);
true
gap> IsMutableDigraph(D);
true
\end{verbatim}

\begin{verbatim}
3.3.3 InducedSubdigraph
\> InducedSubdigraph(digraph, verts)

(\textit{operation})

\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 \texttt{DigraphVertexLabels (5.1.9)}.

If \textit{digraph} belongs to \texttt{IsMutableDigraph (3.1.2)}, then \textit{digraph} is modified in place. If \textit{digraph} belongs to \texttt{IsImmutableDigraph (3.1.3)}, a new immutable digraph containing the appropriate vertices and edges is returned.

\begin{verbatim}
Example
\begin{verbatim}
\end{verbatim}
\end{verbatim}
<mutable multidigraph with 3 vertices, 9 edges>

\[ \text{gap} \rightarrow D = \text{new}; \]
true

3.3.4 ReducedDigraph

- ReducedDigraph\((\text{digraph})\) (operation)
- ReducedDigraphAttr\((\text{digraph})\) (attribute)

**Returns:** A digraph.

This function returns a digraph isomorphic to the subdigraph of \(\text{digraph}\) induced by the set of non-isolated vertices, i.e. the set of those vertices of \(\text{digraph}\) which are the source or range of some edge in \(\text{digraph}\). See InducedSubdigraph (3.3.3).

The ordering of the remaining vertices of \(\text{digraph}\) is preserved, as are the labels of the remaining vertices and edges; see DigraphVertexLabels (5.1.9) and DigraphEdgeLabels (5.1.11). This can allow one to match a vertex in the reduced digraph to the corresponding vertex in \(\text{digraph}\).

If \(\text{digraph}\) is immutable, then a new immutable digraph is returned. Otherwise, the isolated vertices of the mutable digraph \(\text{digraph}\) are removed in-place.

\[ \text{Example} \]

\[ \text{gap} \rightarrow D := \text{Digraph}([ [1, 2], [ ], [ ], [1, 4], [ ] ]); \]<immutable digraph with 5 vertices, 4 edges>
\[ \text{gap} \rightarrow R := \text{ReducedDigraph}(D); \]<immutable digraph with 3 vertices, 4 edges>
\[ \text{gap} \rightarrow \text{OutNeighbours}(R); \]
\[ [ [ 1, 2 ], [ ], [ 1, 3 ] ] \]
\[ \text{gap} \rightarrow \text{DigraphEdges}(D); \]
\[ [ [ 1, 1 ], [ 1, 2 ], [ 4, 1 ], [ 4, 4 ] ] \]
\[ \text{gap} \rightarrow \text{DigraphEdges}(R); \]
\[ [ [ 1, 1 ], [ 1, 2 ], [ 3, 1 ], [ 3, 3 ] ] \]
\[ \text{gap} \rightarrow \text{DigraphVertexLabel}(R, 3); \]
\[ 4 \]
\[ \text{gap} \rightarrow \text{DigraphVertexLabel}(R, 2); \]
\[ 2 \]
\[ \text{gap} \rightarrow D := \text{Digraph}(\text{IsMutableDigraph}, [ [ ], [3], [2] ]); \]<mutable digraph with 3 vertices, 2 edges>
\[ \text{gap} \rightarrow \text{ReducedDigraph}(D); \]<mutable digraph with 2 vertices, 2 edges>

3.3.5 MaximalSymmetricSubdigraph

- MaximalSymmetricSubdigraph\((\text{digraph})\) (operation)
- MaximalSymmetricSubdigraphAttr\((\text{digraph})\) (attribute)
- MaximalSymmetricSubdigraphWithoutLoops\((\text{digraph})\) (operation)
- MaximalSymmetricSubdigraphWithoutLoopsAttr\((\text{digraph})\) (attribute)

**Returns:** A digraph.

If \(\text{digraph}\) is a digraph, then MaximalSymmetricSubdigraph returns a symmetric digraph without multiple edges which has the same vertex set as \(\text{digraph}\), 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.

If \(\text{digraph}\) is immutable, then a new immutable digraph is returned. Otherwise, the mutable digraph \(\text{digraph}\) is changed in-place into such a digraph described above.

See IsSymmetricDigraph (6.1.12), IsMultiDigraph (6.1.10), and DigraphHasLoops (6.1.1) for more information.

**Example**

```gap
gap> D := Digraph([[2, 2], [1, 3], [4], [3, 1]]);
<immutable multidigraph with 4 vertices, 7 edges>
gap> not IsSymmetricDigraph(D) and IsMultiDigraph(D);
true
gap> OutNeighbours(D);
[ [ 2, 2 ], [ 1, 3 ], [ 4 ], [ 3, 1 ] ]
gap> S := MaximalSymmetricSubdigraph(D);
<immutable symmetric digraph with 4 vertices, 4 edges>
gap> IsSymmetricDigraph(S) and not IsMultiDigraph(S);
true
gap> OutNeighbours(S);
[ [ 2 ], [ 1 ], [ 4 ], [ 3 ] ]
gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
gap> MaximalSymmetricSubdigraph(D);
<mutable empty digraph with 3 vertices>
gap> D;
<mutable empty digraph with 3 vertices>
```

### 3.3.6 MaximalAntiSymmetricSubdigraph

\[\text{MaximalAntiSymmetricSubdigraph(digraph)}\] (operation)
\[\text{MaximalAntiSymmetricSubdigraphAttr(digraph)}\] (attribute)

**Returns:** A digraph.

If \(\text{digraph}\) is a digraph, then MaximalAntiSymmetricSubdigraph returns an anti-symmetric subdigraph of \(\text{digraph}\) formed by retaining the vertices of \(\text{digraph}\), discarding any duplicate edges, and discarding any edge \([i, j]\) of \(\text{digraph}\) where \(i > j\) and the reverse edge \([j, i]\) is an edge of \(\text{digraph}\). In other words, for every symmetric pair of edges \([i, j]\) and \([j, i]\) in \(\text{digraph}\), where \(i\) and \(j\) are distinct, it discards the edge \([\max(i, j), \min(i, j)]\).

If \(\text{digraph}\) is immutable, then a new immutable digraph is returned. Otherwise, the mutable digraph \(\text{digraph}\) is changed in-place.

See IsAntiSymmetricDigraph (6.1.2) for more information.

**Example**

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

3.3.7 UndirectedSpanningForest

UndirectedSpanningForest(digraph) (operation)
UndirectedSpanningForestAttr(digraph) (attribute)
UndirectedSpanningTree(digraph) (operation)
UndirectedSpanningTreeAttr(digraph) (attribute)

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 (3.3.5) 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.

If digraph is immutable, then an immutable digraph is returned. Otherwise, the mutable digraph digraph is changed in-place into an undirected spanning tree of digraph.

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

Example

```gap
gap> D := Digraph([[1, 2, 3, 1], [1, 4], [3, 4, 3]]);
<immutable multidigraph with 4 vertices, 9 edges>
gap> UndirectedSpanningTree(D);
fail
gap> forest := UndirectedSpanningForest(D);
<immutable symmetric digraph with 4 vertices, 4 edges>
gap> OutNeighbours(forest);
[ [ 2 ], [ 1 ], [ 4 ], [ 3 ] ]
gap> IsUndirectedSpanningForest(D, forest);
true
gap> DigraphConnectedComponents(forest).comps;
[ [ 2 ], [ 3, 4 ] ]
gap> DigraphConnectedComponents(MaximalSymmetricSubdigraph(D)).comps;
[ [ 1, 2 ], [ 3, 4 ] ]
gap> UndirectedSpanningForest(MaximalSymmetricSubdigraph(D)) > = forest;
true
gap> D := CompleteDigraph(4);
<immutable complete digraph with 4 vertices>
gap> tree := UndirectedSpanningTree(D);
```
3.3.8 QuotientDigraph

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

\textbf{Returns:} A digraph.

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

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

The vertices of \textit{digraph} in part \textit{i} of \textit{p} will become vertex \textit{i} in the quotient, and there exists some edge in \textit{digraph} with source in part \textit{i} and range in part \textit{j} if and only if there is an edge from \textit{i} to \textit{j} in the quotient. In particular, this means that the quotient of a digraph has no multiple edges. which was a change introduced in version 1.0.0 of the \textit{Digraphs} package.

If \textit{digraph} belongs to \textit{IsMutableDigraph} (3.1.2), then \textit{digraph} is modified in place. If \textit{digraph} belongs to \textit{IsImmutableDigraph} (3.1.3), a new immutable digraph with the above properties is returned.

\textbf{Example}

\texttt{gap> D := Digraph([2, 1], [4], [1], [1, 3, 4]);}
\texttt{<immutable digraph with 4 vertices, 7 edges>}
\texttt{gap> DigraphVertices(D);}
\texttt{[1 .. 4]}
\texttt{gap> DigraphEdges(D);}
\texttt{[[1, 2], [1, 1], [2, 4], [3, 1], [4, 1], [4, 3], [4, 4]]}
\texttt{gap> p := [[1], [2, 4], [3]];}
\texttt{[[1], [2, 4], [3]]}
\texttt{gap> quo := QuotientDigraph(D, p);}
\texttt{<immutable digraph with 3 vertices, 6 edges>}
\texttt{gap> DigraphVertices(quo);}
\texttt{[1 .. 3]}
\texttt{gap> DigraphEdges(quo);}
\texttt{[[1, 1], [1, 2], [2, 1], [2, 2], [2, 3], [3, 1]]}
\texttt{gap> QuotientDigraph(EmptyDigraph(0), []);}
\texttt{<immutable empty digraph with 0 vertices>
3.3.9 DigraphReverse

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

(operation)

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

(attribute)

**Returns:** A digraph.

The reverse of a digraph is the digraph formed by reversing the orientation of each of its edges, i.e. for every edge \([i, j]\) of a digraph, the reverse contains the corresponding edge \([j, i]\).

`DigraphReverse` returns the reverse of the digraph `digraph`. If `digraph` is immutable, then a new immutable digraph is returned. Otherwise, the mutable digraph `digraph` is changed in-place into its reverse.

```gap
gap> D := Digraph([[3], [1, 3, 5], [1], [1, 2, 4], [2, 3, 5]]);
<immutable digraph with 5 vertices, 11 edges>
gap> DigraphReverse(D);
<immutable digraph with 5 vertices, 11 edges>
gap> OutNeighbours(last);
[ [ 2, 3, 4 ], [ 4, 5 ], [ 1, 2, 5 ], [ 4 ], [ 2, 5 ] ]
gap> D := Digraph([[2, 4], [1], [4], [3, 4]]);
<immutable digraph with 4 vertices, 6 edges>
gap> DigraphEdges(D);  
[ [ 1, 2 ], [ 1, 4 ], [ 2, 1 ], [ 3, 4 ], [ 4, 3 ], [ 4, 4 ] ]
gap> DigraphEdges(DigraphReverse(D));
[ [ 1, 2 ], [ 2, 1 ], [ 3, 4 ], [ 4, 1 ], [ 4, 3 ], [ 4, 4 ] ]
gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
gap> DigraphEdges(D);
[ [ 2 ], [ 3 ], [ 1 ] ]
gap> DigraphReverse(D);
<mutable digraph with 3 vertices, 3 edges>
gap> OutNeighbours(D);
[ [ 3 ], [ 1 ], [ 2 ] ]
```

3.3.10 DigraphDual

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

(operation)

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

(attribute)

**Returns:** A digraph.

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`. The dual is sometimes called the complement.

`DigraphDual` returns the dual of the digraph `digraph`. If `digraph` is an immutable digraph, then a new immutable digraph is returned. Otherwise, the mutable digraph `digraph` is changed in-place into its dual.

```gap
gap> D := Digraph([[3], [1, 3, 5], [1], [1, 2, 4], [2, 3, 5]]);
<immutable digraph with 5 vertices, 11 edges>
gap> DigraphDual(D);  
<immutable digraph with 5 vertices, 11 edges>
gap> OutNeighbours(last);
[ [ 2, 3, 4 ], [ 4, 5 ], [ 1, 2, 5 ], [ 4 ], [ 2, 5 ] ]
gap> D := Digraph([[2, 4], [1], [4], [3, 4]]);
<immutable digraph with 4 vertices, 6 edges>
gap> DigraphEdges(D);  
[ [ 1, 2 ], [ 1, 4 ], [ 2, 1 ], [ 3, 4 ], [ 4, 3 ], [ 4, 4 ] ]
gap> DigraphEdges(DigraphDual(D));
[ [ 1, 2 ], [ 2, 1 ], [ 3, 4 ], [ 4, 1 ], [ 4, 3 ], [ 4, 4 ] ]
gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
gap> DigraphDual(D);
<mutable digraph with 3 vertices, 3 edges>
gap> OutNeighbours(D);
[ [ 3 ], [ 1 ], [ 2 ] ]
```
3.3.11 DigraphSymmetricClosure

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

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

**Returns:** A digraph.

If \text{digraph} is a digraph, then this attribute gives the minimal symmetric digraph which has the same vertices and contains all the edges of \text{digraph}.

A digraph is **symmetric** if its adjacency matrix \text{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 \(u\) to a vertex \(v\) as there are from \(v\) to \(u\); see \text{IsSymmetricDigraph} (6.1.12).

If \text{digraph} is immutable, then a new immutable digraph is returned. Otherwise, the mutable digraph \text{digraph} is changed in-place into its symmetric closure.

Example

\[ \text{gap} > D := \text{Digraph}([[1, 2, 3], [2, 4], [1], [3, 4]]); \]
\[ <\text{immutable digraph with 4 vertices, 8 edges}> \]
\[ \text{gap} > D := \text{DigraphSymmetricClosure}(D); \]
\[ <\text{immutable symmetric digraph with 4 vertices, 11 edges}> \]
\[ \text{gap} > \text{IsSymmetricDigraph}(D); \]
\[ \text{true} \]
\[ \text{gap} > \text{List}(<\text{OutNeighbours}(D), \text{AsSet}>); \]
\[ [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 4 ], [ 2, 3, 4 ] ] \]
\[ \text{gap} > D := \text{Digraph}([[2, 2], [1]]); \]
\[ <\text{immutable multidigraph with 2 vertices, 3 edges}> \]
\[ \text{gap} > D := \text{DigraphSymmetricClosure}(D); \]
\[ <\text{immutable symmetric multidigraph with 2 vertices, 4 edges}> \]
\[ \text{gap} > \text{OutNeighbours}(D); \]
\[ [ [ 2, 2 ], [ 1, 1 ] ] \]
\[ \text{gap} > D := \text{CycleDigraph}(\text{IsMutableDigraph}, 3); \]
\[ <\text{mutable digraph with 3 vertices, 3 edges}> \]
\[ \text{gap} > \text{DigraphSymmetricClosure}(D); \]
\[ <\text{mutable digraph with 3 vertices, 6 edges}> \]
\[ \text{gap} > D; \]
\[ <\text{mutable digraph with 3 vertices, 6 edges}> \]

3.3.12 DigraphTransitiveClosure

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

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

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

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

**Returns:** A digraph.

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

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

A **digraph** is the least (reflexive and) transitive digraph containing *digraph*.

If *digraph* is immutable, then a new immutable digraph is returned. Otherwise, the mutable digraph *digraph* is changed in-place into its (reflexive) transitive closure.

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

**Example**
```gap
gap> D := DigraphFromDiSparse6String(".H'eOWR'Ul^");
<immutable digraph with 9 vertices, 8 edges>

gap> IsReflexiveDigraph(D) or IsTransitiveDigraph(D);
falset

gap> OutNeighbours(D);
[ [ 4, 6 ], [ 1, 3 ], [ ], [ 5 ], [ ], [ 7, 8, 9 ], [ ], [ ], [ ] ]

gap> T := DigraphTransitiveClosure(D);
<immutable transitive digraph with 9 vertices, 18 edges>

gap> OutNeighbours(T);
[ [ 4, 6, 5, 7, 8, 9 ], [ 1, 3, 4, 5, 6, 7, 8, 9 ], [ ], [ 5 ],
  [ ], [ 7, 8, 9 ], [ ], [ ], [ ] ]

gap> RT := DigraphReflexiveTransitiveClosure(D);
<immutable preorder digraph with 9 vertices, 27 edges>

gap> OutNeighbours(RT);
[ [ 4, 6, 5, 7, 8, 9, 1 ], [ 1, 3, 4, 5, 6, 7, 8, 9, 2 ], [ 3 ],
  [ 5, 4 ], [ 5 ], [ 7, 8, 9, 6 ], [ 7 ], [ 8 ], [ 9 ] ]

gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>

gap> DigraphReflexiveTransitiveClosure(D);
<mutable digraph with 3 vertices, 9 edges>

gap> D;
<mutable digraph with 3 vertices, 9 edges>
```

### 3.3.13 DigraphTransitiveReduction

**DigraphTransitiveReduction**

**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 (3.3.12)]. In order words, it is the least subgraph of *digraph* which retains the same reachability as *digraph*.

If *digraph* is immutable, then a new immutable digraph is returned. Otherwise, the mutable digraph *digraph* is changed in-place into its (reflexive) transitive reduction.

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
D := Digraph([[1, 2, 3], [3], [3]]);
true
D1 := DigraphReflexiveTransitiveReduction(D);
<immutable digraph with 3 vertices, 2 edges>
DigraphHasLoops(D1);
false
OutNeighbours(D1);
[ [ 2 ], [ 3 ], [ ] ]
D2 := DigraphTransitiveReduction(D);
<immutable digraph with 3 vertices, 4 edges>
DigraphHasLoops(D2);
true
OutNeighbours(D2);
[ [ 2, 1 ], [ 3 ], [ 3 ] ]
DigraphReflexiveTransitiveClosure(D)
> = DigraphReflexiveTransitiveClosure(D1);
true
DigraphTransitiveClosure(D)
> = DigraphTransitiveClosure(D2);
true
D := Digraph(IsMutableDigraph, [[1], [1], [1, 2, 3]]);
<mutable digraph with 3 vertices, 5 edges>
DigraphReflexiveTransitiveReduction(D);
<mutable digraph with 3 vertices, 2 edges>
```

### 3.3.14 DigraphAddVertex

> **DigraphAddVertex**(digraph[, label])

    (operation)

**Returns:** A digraph.

The operation returns a digraph constructed from `digraph` by adding a single new vertex, and no new edges.

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

If `digraph` belongs to IsMutableDigraph (3.1.2), then the vertex is added directly to `digraph`. If `digraph` belongs to IsImmutableDigraph (3.1.3), an immutable copy of `digraph` with the additional vertex is returned.

```gap
D := CompleteDigraph(3);
<immutable complete digraph with 3 vertices>
new := DigraphAddVertex(D);
<immutable digraph with 4 vertices, 6 edges>
D = new;
false
DigraphVertices(new);
[ 1 .. 4 ]
ew := DigraphAddVertex(D, Group([[1, 2]]));
<immutable digraph with 4 vertices, 6 edges>
```
Digraphs

```
gap> DigraphVertexLabels(new);
[ 1, 2, 3, Group([ (1,2) ]) ]
gap> D := CompleteBipartiteDigraph(IsMutableDigraph, 2, 3);
<mutable digraph with 5 vertices, 12 edges>
gap> new := DigraphAddVertex(D);
<mutable digraph with 6 vertices, 12 edges>
gap> D = new;
true
```

### 3.3.15 DigraphAddVertices (for a digraph and an integer)

**Operation**

```
DigraphAddVertices(digraph, m)
```

**Returns:**

A digraph.

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

Otherwise, if `labels` is a list consisting of $k$ GAP objects, then this operation returns a digraph constructed from `digraph` by adding $k$ new vertices, which are labelled according to this list.

If `digraph` belongs to `IsMutableDigraph (3.1.2)`, then the vertices are added directly to `digraph`, which is changed in-place. If `digraph` belongs to `IsImmutableDigraph (3.1.3)`, then the result is returned as an immutable digraph.

**Example**

```
gap> D := CompleteDigraph(3);
<immutable complete digraph with 3 vertices>
gap> new := DigraphAddVertices(D, 3);
<immutable digraph with 6 vertices, 6 edges>
gap> DigraphVertices(new);
[ 1 .. 6 ]
gap> new := DigraphAddVertices(D, [Group([[1, 2]]), "d"]);
<immutable digraph with 5 vertices, 6 edges>
gap> DigraphVertexLabels(new);
[ 1, 2, 3, Group([ (1,2) ]), "d" ]
gap> DigraphAddVertices(D, 0) = D;
true
gap> D := CompleteBipartiteDigraph(IsMutableDigraph, 2, 3);
<mutable digraph with 5 vertices, 12 edges>
gap> new := DigraphAddVertices(D, 4);
<mutable digraph with 9 vertices, 12 edges>
gap> D = new;
true
```

### 3.3.16 DigraphAddEdge (for a digraph and an edge)

**Operation**

```
DigraphAddEdge(digraph, edge)
```

**Returns:**

A digraph.

If `edge` is a pair of vertices of `digraph`, or `src` and `ran` are vertices of `digraph`, then this operation returns a digraph constructed from `digraph` by adding a new edge with source `edge[1] [src]` and range `edge[2] [ran]`.

```
gap> D := CompleteDigraph(3);
<immutable complete digraph with 3 vertices>
gap> new := DigraphAddVertices(D, 3);
<immutable digraph with 6 vertices, 6 edges>
```
If `digraph` belongs to `IsMutableDigraph (3.1.2)`, then the edge is added directly to `digraph`. If `digraph` belongs to `IsImmutableDigraph (3.1.3)`, then an immutable copy of `digraph` with the additional edge is returned.

```gap
D1 := Digraph([[2], [3], []]);
<immutable digraph with 3 vertices, 2 edges>
DigraphEdges(D1);
[ [ 1, 2 ], [ 2, 3 ] ]
D2 := DigraphAddEdge(D1, [3, 1]);
<immutable digraph with 3 vertices, 3 edges>
DigraphEdges(D2);
[ [ 1, 2 ], [ 2, 3 ], [ 3, 1 ] ]
D3 := DigraphAddEdge(D2, [2, 3]);
<immutable multidigraph with 3 vertices, 4 edges>
DigraphEdges(D3);
[ [ 1, 2 ], [ 2, 3 ], [ 2, 3 ], [ 3, 1 ] ]
D := CycleDigraph(IsMutableDigraph, 4);
<mutable digraph with 4 vertices, 4 edges>
new := DigraphAddEdge(D, [1, 3]);
<mutable digraph with 4 vertices, 5 edges>
DigraphEdges(new);
[ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ], [ 3, 4 ], [ 4, 1 ] ]
D = new;
true
```

### Example

<table>
<thead>
<tr>
<th>DigraphAddEdgeOrbit</th>
</tr>
</thead>
</table>

⊿ `DigraphAddEdgeOrbit(digraph, edge)`  

(operation)

**Returns:** A new digraph.

This operation returns a new digraph with the same vertices and edges as `digraph` and with additional edges consisting of the orbit of the edge `edge` under the action of the `DigraphGroup (7.2.10)` of `digraph`. If `edge` is already an edge in `digraph`, then `digraph` is returned unchanged. The argument `digraph` must be an immutable digraph.

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

```gap
gr1 := CayleyDigraph(DihedralGroup(8));
<immutable digraph with 8 vertices, 24 edges>
gr2 := DigraphAddEdgeOrbit(gr1, [1, 8]);
<immutable digraph with 8 vertices, 32 edges>
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 ] ]
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 ] ]
```
3.3.18 DigraphAddEdges

```
DigraphAddEdges(digraph, edges)
```

Returns: A digraph.

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

If an edge is included in `edges` with multiplicity k, then it will be added k times. If `digraph` belongs to IsMutableDigraph (3.1.2), then the edges are added directly to `digraph`. If `digraph` belongs to IsImmutableDigraph (3.1.3), then the result is returned as an immutable digraph.

Example

```
gap> func := function(n)
gap>   local source, range, i;
gap>   source := [];
gap>   range := [];
gap>   for i in [1 .. n - 2] do
gap>     Add(source, i);
gap>     Add(range, i + 1);
gap>   od;
gap>   return Digraph(n, source, range);
gap> end;;
gap> D := func(1024);
gap> new := DigraphAddEdges(D, 
               [[1023, 1024], [1, 1024], [1023, 1024], [1024, 1]]);
gap> D = new;
gap> false
```

3.3.19 DigraphRemoveVertex

```
DigraphRemoveVertex(digraph, v)
```

Returns: A digraph.

If `v` is a vertex of `digraph`, then this operation returns a 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 returned digraph are [1 .. n-1], but the original labels can be accessed via DigraphVertexLabels (5.1.9).

Example

```
gap> gr3 := DigraphRemoveEdgeOrbit(gr2, [1, 8]);
gap> gr3 = gr1;
gap> true
```
If \( \text{digraph} \) belongs to \( \text{IsMutableDigraph} \) (3.1.2), then the vertex is removed directly from \( \text{digraph} \). If \( \text{digraph} \) belongs to \( \text{IsImmutableDigraph} \) (3.1.3), an immutable copy of \( \text{digraph} \) without the vertex is returned.

```gap
gap> D := Digraph(["a", "b", "c"],
>                ["a", "a", "b", "c", "c"],
>                ["b", "c", "a", "a", "c"]);
<immutable digraph with 3 vertices, 5 edges>
gap> DigraphVertexLabels(D);
[ "a", "b", "c" ]
gap> DigraphEdges(D);
[ [ 1, 2 ], [ 1, 3 ], [ 2, 1 ], [ 3, 1 ], [ 3, 3 ] ]
gap> new := DigraphRemoveVertex(D, 2);
<immutable digraph with 2 vertices, 3 edges>
gap> DigraphVertexLabels(new);
[ "a", "c" ]
gap> D := CycleDigraph(IsMutableDigraph, 5);
<mutable digraph with 5 vertices, 5 edges>
gap> new := DigraphRemoveVertex(D, 1);
<mutable digraph with 4 vertices, 3 edges>
gap> DigraphVertexLabels(D);
[ 2, 3, 4, 5 ]
gap> D = new;
true
```

### 3.3.20 DigraphRemoveVertices

\( \text{DigraphRemoveVertices}(\text{digraph}, \text{verts}) \)  
(operation)

**Returns:** A digraph.

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

If \( \text{digraph} \) has \( n \) vertices, then the vertices of the new digraph are \([1 \ldots n - \text{Length}(\text{verts})]\), but the original labels can be accessed via \( \text{DigraphVertexLabels} \) (5.1.9).

If \( \text{digraph} \) belongs to \( \text{IsMutableDigraph} \) (3.1.2), then the vertices are removed directly from \( \text{digraph} \). If \( \text{digraph} \) belongs to \( \text{IsImmutableDigraph} \) (3.1.3), an immutable copy of \( \text{digraph} \) without the vertices is returned.

```gap
gap> D := Digraph([[3], [1, 3, 5], [1], [1, 2, 4], [2, 3, 5]]);
<immutable digraph with 5 vertices, 11 edges>
gap> SetDigraphVertexLabels(D, ["a", "b", "c", "d", "e"]);
gap> new := DigraphRemoveVertices(D, [2, 4]);
<immutable digraph with 3 vertices, 4 edges>
gap> DigraphVertexLabels(new);
[ "a", "c", "e" ]
gap> D := CycleDigraph(IsMutableDigraph, 5);
<mutable digraph with 5 vertices, 5 edges>
gap> new := DigraphRemoveVertices(D, [1, 3]);
<mutable digraph with 3 vertices, 1 edge>
gap> DigraphVertexLabels(D);
[ 2, 4, 5 ]
```
### 3.3.21 DigraphRemoveEdge (for a digraph and an edge)

- **DigraphRemoveEdge(digraph, edge)**
  - (operation)

  **Returns:** A digraph.

  If `digraph` is a digraph with no multiple edges and `edge` is a pair of vertices of `digraph`, or `src` and `ran` are vertices of `digraph`, then this operation returns a digraph constructed from `digraph` by removing the edge specified by `edge` or `[src, ran]`.

  If `digraph` belongs to `IsMutableDigraph` (3.1.2), then the edge is removed directly from `digraph`. If `digraph` belongs to `IsImmutableDigraph` (3.1.3), an immutable copy of `digraph` without the edge is returned.

  Note that if `digraph` belongs to `IsImmutableDigraph` (3.1.3), then a new copy of `digraph` will be returned even if `edge` or `[src, ran]` does not define an edge of `digraph`.

- **Example**

```gap
gap> D := CycleDigraph(250000);
<immutable cycle digraph with 250000 vertices>
gap> D := DigraphRemoveEdge(D, [250000, 1]);
<immutable digraph with 250000 vertices, 249999 edges>
gap> new := DigraphRemoveEdge(D, [25000, 2]);;
gap> new = D;
true
gap> IsIdenticalObj(new, D);
false
gap> D := DigraphMutableCopy(D);;
gap> new := DigraphRemoveEdge(D, 2500, 2);;
gap> IsIdenticalObj(new, D);  
true
```

### 3.3.22 DigraphRemoveEdgeOrbit

- **DigraphRemoveEdgeOrbit(digraph, edge)**
  - (operation)

  **Returns:** A new digraph.

  This operation returns a new digraph with the same vertices as `digraph` and with the orbit of the edge `edge` (under the action of the `DigraphGroup` (7.2.10) of `digraph`) removed. If `edge` is not an edge in `digraph`, then `digraph` is returned unchanged. The argument `digraph` must be an immutable digraph.

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

- **Example**

```gap
gap> gr1 := CayleyDigraph(DihedralGroup(8));
<immutable digraph with 8 vertices, 24 edges>
gap> gr2 := DigraphAddEdgeOrbit(gr1, [1, 8]);
<immutable 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 ],
```
Digraphs

[ 7, 8 ], [ 7, 1 ], [ 7, 3 ], [ 8, 7 ], [ 8, 6 ], [ 8, 5 ]

\[
gap\text{DigraphEdges(gr2);} \\\text{[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 8 ], [ 2, 1 ], [ 2, 8 ], [ 2, 6 ], [ 2, 3 ], [ 3, 5 ], [ 3, 4 ], [ 3, 7 ], [ 3, 2 ], [ 4, 6 ], [ 4, 7 ], [ 4, 1 ], [ 4, 5 ], [ 5, 3 ], [ 5, 2 ], [ 5, 8 ], [ 5, 4 ], [ 6, 4 ], [ 6, 5 ], [ 6, 2 ], [ 6, 7 ], [ 7, 8 ], [ 7, 1 ], [ 7, 3 ], [ 7, 6 ], [ 8, 7 ], [ 8, 6 ], [ 8, 5 ], [ 8, 1 ] \]

\[
gap\text{gr3 := DigraphRemoveEdgeOrbit(gr2, [1, 8])}; \\text{<immutable digraph with 8 vertices, 24 edges>}
gap\text{gr3 = gr1;} \\text{true}
\]

3.3.23 DigraphRemoveEdges

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

\[
\text{Returns: A digraph.}
\]

\[
\text{If one of the following holds:}
\]

\[
\begin{itemize}
\item digraph is a digraph with no multiple edges, and edges is a list of pairs of vertices of digraph, or
\item digraph is a digraph and edges is an empty list
\end{itemize}
\]

\[
\text{then this operation returns a digraph constructed from digraph by removing all of the edges specified by edges (see DigraphRemoveEdge (3.3.21)).}
\]

\[
\text{If digraph belongs to IsMutableDigraph (3.1.2), then the edge is removed directly from digraph. If digraph belongs to IsImmutableDigraph (3.1.3), the edge is removed from an immutable copy of digraph and this new digraph is returned.}
\]

\[
\text{Note that if edges is empty, then this operation will always return digraph rather than a copy. Also, if any element of edges is invalid (i.e. does not define an edge of digraph) then that element will simply be ignored.}
\]

\[
\text{Example}
\]

\[
\text{gap} \text{D := CycleDigraph(250000);} \text{<immutable cycle digraph with 250000 vertices>}
gap\text{D := DigraphRemoveEdges(D, [[250000, 1]]);} \text{<immutable digraph with 250000 vertices, 249999 edges>}
gap\text{D := DigraphMutableCopy(D);} \text{<mutable digraph with 250000 vertices, 249999 edges>}
gap\text{new := DigraphRemoveEdges(D, [[1, 2], [2, 3], [3, 100]]);} \text{<mutable digraph with 250000 vertices, 249997 edges>}
gap\text{new = D;} \text{true}
\]

3.3.24 DigraphRemoveLoops

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

\[
\text{DigraphRemoveLoopsAttr(digraph)} \quad (\text{attribute})
\]

\[
\text{Returns: A digraph.}
\]
If \textit{digraph} is a digraph, then this operation returns a digraph constructed from \textit{digraph} by removing every loop. A loop is an edge with equal source and range.

If \textit{digraph} is immutable, then a new immutable digraph is returned. Otherwise, the loops are removed from the mutable digraph \textit{digraph} in-place.

\begin{verbatim}
Example

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

Example

D := Digraph(IsMutableDigraph, [[1, 2], [1]]);
<mutable digraph with 2 vertices, 3 edges>
D := DigraphRemoveLoops(D);
<mutable digraph with 2 vertices, 2 edges>
D;
<mutable digraph with 2 vertices, 2 edges>

3.3.25 \textbf{DigraphRemoveAllMultipleEdges}

\begin{verbatim}
\textbf{DigraphRemoveAllMultipleEdges} (digraph)
\textbf{DigraphRemoveAllMultipleEdgesAttr} (digraph)
\end{verbatim}

\textbf{Returns:} A digraph.

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

If \textit{digraph} is immutable, then a new immutable digraph is returned. Otherwise, the multiple edges of the mutable digraph \textit{digraph} are removed in-place.

\begin{verbatim}
Example

D1 := Digraph([[1, 2, 3, 2], [1, 1, 3], [2, 2, 2]]);
<immutable multidigraph with 3 vertices, 10 edges>
D2 := DigraphRemoveAllMultipleEdges(D1);
<immutable digraph with 3 vertices, 6 edges>
OutNeighbours(D2);
[ [ 1, 2, 3 ], [ 1, 3 ], [ 2 ] ]
D := DigraphRemoveAllMultipleEdgesAttr(D1);
<mutable multidigraph with 2 vertices, 3 edges>
D := DigraphRemoveAllMultipleEdges(D);
<mutable digraph with 2 vertices, 2 edges>
D;
<mutable digraph with 2 vertices, 2 edges>

3.3.26 \textbf{DigraphReverseEdges (for a digraph and a list of edges)}

\begin{verbatim}
\textbf{DigraphReverseEdges} (digraph, edges)
\textbf{DigraphReverseEdge} (digraph, edge)
\textbf{DigraphReverseEdge} (digraph, src, ran)
\end{verbatim}

\textbf{Returns:} A digraph.

If \textit{digraph} is a digraph without multiple edges, and \textit{edges} is 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), then \textbf{DigraphReverseEdges} returns a digraph constructed from \textit{digraph} by reversing the orientation of
every edge specified by `edges`. If only one edge is to be reversed, then `DigraphReverseEdge` can be used instead. In this case, the second argument should just be a single vertex-pair, or the second and third arguments should be the source and range of an edge respectively.

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

If `digraph` belongs to `IsMutableDigraph (3.1.2)`, then the edges are reversed in `digraph`. If `digraph` belongs to `IsImmutableDigraph (3.1.3)`, an immutable copy of `digraph` with the specified edges reversed is returned.

```gap
gap> D := DigraphFromDiSparse6String(".Tg?i@s?t_e?_qEsC"); <immutable digraph with 21 vertices, 8 edges>
gap> DigraphEdges(D);
[ [ 1, 2 ], [ 1, 7 ], [ 1, 8 ], [ 5, 21 ], [ 7, 19 ], [ 9, 1 ],
  [ 11, 2 ], [ 21, 1 ] ]
gap> new := DigraphReverseEdge(D, [7, 19]); <immutable digraph with 21 vertices, 8 edges>
gap> DigraphEdges(new);
[ [ 1, 2 ], [ 1, 7 ], [ 1, 8 ], [ 5, 21 ], [ 9, 1 ], [ 11, 2 ],
  [ 19, 7 ], [ 21, 1 ] ]
gap> D2 := DigraphMutableCopy(new);;
gap> new := DigraphReverseEdges(D2, [[19, 7]]);;
gap> D2 = new;
true
gap> D = new;
true
```

### 3.3.27 DigraphDisjointUnion (for an arbitrary number of digraphs)

- `DigraphDisjointUnion(D1, D2, ...)`
- `DigraphDisjointUnion(list)`

**Returns:** A digraph.

In the first form, if `D1, D2, etc. are digraphs, then DigraphDisjointUnion returns their disjoint union. In the second form, if `list` is a non-empty list of digraphs, then DigraphDisjointUnion returns the disjoint union of the digraphs contained in the list.

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

More specifically, for a collection of digraphs `D1, D2, ...,` the disjoint union with have `DigraphNrVertices(D1) + DigraphNrVertices(D2) + ...` vertices. The edges of `D1` will remain unchanged, whilst the edges of the `i`th digraph, `D[i]`, will be changed so that they belong to the vertices of the disjoint union corresponding to `D[i]`. In particular, the edges of `D[i]` will have their source and range increased by `DigraphNrVertices(D1) + ... + DigraphNrVertices(D[i-1])`.

Note that previously set `DigraphVertexLabels (5.1.9)` will be lost.

If the first digraph `D1 [list[1]]` belongs to `IsMutableDigraph (3.1.2)`, then `D1 [list[1]]` is modified in place to contain the appropriate vertices and edges. If `digraph` belongs to `IsImmutableDigraph (3.1.3)`, a new immutable digraph containing the appropriate vertices and edges is returned.

```gap
gap> D1 := CycleDigraph(3);
<immutable cycle digraph with 3 vertices>
gap> OutNeighbours(D1);
```
Digraphs

[ 2 ], [ 3 ], [ 1 ]

\textbf{Example}

\begin{verbatim}
gap> D := CompleteDigraph(3);  
<immutable complete digraph with 3 vertices>
\gap> OutNeighbours(D2);  
[ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ]
\gap> union := DigraphDisjointUnion(D1, D2);  
<immutable digraph with 6 vertices, 9 edges>
\gap> OutNeighbours(union);  
[ [ 2 ], [ 3 ], [ 1 ], [ 5, 6 ], [ 4, 6 ], [ 4, 5 ] ]
\end{verbatim}

3.3.28 \textbf{DigraphEdgeUnion (for a positive number of digraphs)}

\begin{itemize}
\item \textbf{DigraphEdgeUnion(D1, D2, ...)} \hspace{1cm} (function)
\item \textbf{DigraphEdgeUnion(list)} \hspace{1cm} (function)
\end{itemize}

\textbf{Returns:} A digraph.

In the first form, if \texttt{D1}, \texttt{D2}, etc. are digraphs, then \texttt{DigraphEdgeUnion} returns their edge union. In the second form, if \texttt{list} is a non-empty list of digraphs, then \texttt{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 \texttt{DigraphVertexLabels (5.1.9)} will be lost.

If the first digraph \texttt{D1 [list[1]]} belongs to \texttt{IsMutableDigraph (3.1.2)}, then \texttt{D1 [list[1]]} is modified in place to contain the appropriate vertices and edges. If \texttt{digraph} belongs to \texttt{IsImmutableDigraph (3.1.3)}, a new immutable digraph containing the appropriate vertices and edges is returned.

\begin{verbatim}
gap> D := CycleDigraph(10);  
<immutable cycle digraph with 10 vertices>
\gap> DigraphEdgeUnion(D, D);  
<immutable multidigraph with 10 vertices, 20 edges>
\gap> D1 := Digraph([[2], [1]]);  
<immutable digraph with 2 vertices, 2 edges>
\gap> D2 := Digraph([[2, 3], [2], [1]]);  
<immutable digraph with 3 vertices, 4 edges>
\gap> union := DigraphEdgeUnion(D1, D2);  
<immutable multidigraph with 3 vertices, 6 edges>
\gap> OutNeighbours(union);  
[ [ 2, 2, 3 ], [ 1, 2 ], [ 1 ] ]
\gap> union = DigraphByEdges(  
> Concatenation(DigraphEdges(D1), DigraphEdges(D2)));
true
\end{verbatim}

3.3.29 \textbf{DigraphJoin (for a positive number of digraphs)}

\begin{itemize}
\item \textbf{DigraphJoin(D1, D2, ...)} \hspace{1cm} (function)
\item \textbf{DigraphJoin(list)} \hspace{1cm} (function)
\end{itemize}

\textbf{Returns:} A digraph.
In the first form, if $D_1, D_2, \ldots$ are digraphs, then \texttt{DigraphJoin} returns their join. In the second form, if \texttt{list} is a non-empty list of digraphs, then \texttt{DigraphJoin} returns the join of the digraphs contained in the list.

The join of a collection of digraphs $D_1, D_2, \ldots$ is formed by first taking the \texttt{DigraphDisjointUnion} (3.3.27) of the collection. In the disjoint union, if $i \neq j$ then there are no edges between vertices corresponding to digraphs $D[i]$ and $D[j]$ 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 \texttt{DigraphVertexLabels} (5.1.9) will be lost.

If the first digraph $D_1 [\text{list[1]}]$ belongs to \texttt{IsMutableDigraph} (3.1.2), then $D_1 [\text{list[1]}]$ is modified in place to contain the appropriate vertices and edges. If \texttt{digraph} belongs to \texttt{IsImmutableDigraph} (3.1.3), a new immutable digraph containing the appropriate vertices and edges is returned.

Example

\begin{verbatim}
gap> D := CompleteDigraph(3);
<immutable complete digraph with 3 vertices>
gap> IsCompleteDigraph(DighraphJoin(D, D));
true
gap> D2 := CycleDigraph(3);
<immutable cycle digraph with 3 vertices>
gap> DighraphJoin(D, D2);
<immutable digraph with 6 vertices, 27 edges>
\end{verbatim}

3.3.30 \hspace{1em} \textbf{DigraphCartesianProduct} (for a positive number of digraphs)

\begin{verbatim}
> DigraphCartesianProduct(gr1, gr2, ...)
(function)
> DigraphCartesianProduct(list)
(function)
\end{verbatim}

\textbf{Returns:} A digraph.

In the first form, if $gr1, gr2, \ldots$ are digraphs, then \texttt{DigraphCartesianProduct} returns a digraph isomorphic to their Cartesian product.

In the second form, if \texttt{list} is a non-empty list of digraphs, then \texttt{DigraphCartesianProduct} returns a digraph isomorphic to the Cartesian product of the digraphs contained in the list.

Mathematically, the Cartesian product of two digraphs $G, H$ is a digraph with vertex set $\text{Cartesian(DigraphVertices(G), DigraphVertices(H))}$ such that there is an edge from $[u, u']$ to $[v, v']$ iff $u = v$ and there is an edge from $u'$ to $v'$ in $H$ or $u' = v'$ and there is an edge from $u$ to $v$ in $G$.

Due to technical reasons, the digraph $D$ returned by \texttt{DigraphCartesianProduct} has vertex set $[1 .. \text{DigraphNrVertices(G)}*\text{DigraphNrVertices(H)}]$ instead, and the exact method of encoding pairs of vertices into integers is implementation specific. The original vertex pair can be somewhat regained by using \texttt{DigraphCartesianProductProjections} (3.3.32). In addition, \texttt{DigraphVertexLabels} (5.1.9) are preserved: if vertex pair $[u, u']$ was encoded as $i$ then the vertex label of $i$ will be the pair of vertex labels of $u$ and $u'$ i.e. \texttt{DigraphVertexLabel(D,i) = [DigraphVertexLabel(G,u), DigraphVertexLabel(H,u')]}

As the Cartesian product is associative, the Cartesian product of a collection of digraphs $gr1, gr2, \ldots$ is computed in the obvious fashion.

Example

\begin{verbatim}
gap> gr := ChainDigraph(4);
<immutable chain digraph with 4 vertices>
\end{verbatim}
3.3.31 DigraphDirectProduct (for a positive number of digraphs)

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

**Returns:** A digraph.

In the first form, if \( \text{gr1}, \text{gr2}, \text{etc.} \) are digraphs, then \text{DigraphDirectProduct} returns a digraph isomorphic to their direct product.

In the second form, if \( \text{list} \) is a non-empty list of digraphs, then \text{DigraphDirectProduct} returns a digraph isomorphic to the direct product of the digraphs contained in the list.

Mathematically, the direct product of two digraphs \( G, H \) is a digraph with vertex set \( \text{Cartesian(DigraphVertices(G), DigraphVertices(H))} \) such that there is an edge from \([u, u']\) to \([v, v']\) iff there is an edge from \(u\) to \(v\) in \(G\) and an edge from \(u'\) to \(v'\) in \(H\).

Due to technical reasons, the digraph \(D\) returned by \text{DigraphDirectProduct} has vertex set \([1 .. \text{DigraphNrVertices}(G) \ast \text{DigraphNrVertices}(H)]\) instead, and the exact method of encoding pairs of vertices into integers is implementation specific. The original vertex pair can be somewhat regained by using \text{DigraphDirectProductProjections} (3.3.33). In addition \text{DigraphVertexLabels} (5.1.9) are preserved: if vertex pair \([u, u']\) was encoded as \(i\) then the vertex label of \(i\) will be the pair of vertex labels of \(u\) and \(u'\) i.e. \text{DigraphVertexLabel}(D, i) = \text{DigraphVertexLabel}(G, u), \text{DigraphVertexLabel}(H, u').

As the direct product is associative, the direct product of a collection of digraphs \(\text{gr1, gr2, ...}\) is computed in the obvious fashion.

\[ \begin{align*}
\text{gap> gr} & := \text{ChainDigraph}(4); \\
& \quad <\text{immutable chain digraph with 4 vertices}> \\
\text{gap> gr2} & := \text{CycleDigraph}(3); \\
& \quad <\text{immutable cycle digraph with 3 vertices}> \\
\text{gap> gr3} & := \text{DigraphDirectProduct}(\text{gr, gr2}); \\
& \quad <\text{immutable digraph with 12 vertices, 9 edges}> \\
\text{gap> IsIsomorphicDigraph}(& \text{gr3}, \\
& \quad > \text{Digraph}([[6], [7], [8], []], \\
& \quad > \quad [10], [11], [12], []), \\
& \quad > \quad [2], [3], [4], [4]])); \\
\text{true}
\end{align*} \]

3.3.32 DigraphCartesianProductProjections

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

**Returns:** A list of transformations.
If \textit{digraph} is a Cartesian product digraph, \textit{digraph} = \texttt{DigraphCartesianProduct(gr\_1, gr\_2, \ldots )}, then \texttt{DigraphCartesianProductProjections} returns a list \texttt{proj} such that \texttt{proj[i]} is the projection onto the \texttt{i}-th coordinate of the product.

A projection is an idempotent endomorphism of \textit{digraph}. If \texttt{gr\_1}, \texttt{gr\_2}, \ldots are all loopless digraphs, then the induced subdigraph of \textit{digraph} on the image of \texttt{proj[i]} is isomorphic to \texttt{gr\_i}.

Currently this attribute is only set upon creating an immutable digraph via \texttt{DigraphCartesianProduct} and there is no way of calculating it for other digraphs.

For more information see \texttt{DigraphCartesianProduct (3.3.30)}

\begin{verbatim}
gap> D := DigraphCartesianProduct(ChainDigraph(3), CycleDigraph(4), Digraph([[2], [2]]));;
true
gap> HasDigraphCartesianProductProjections(D);
true
gap> proj := DigraphCartesianProductProjections(D);
Length(proj);
3
gap> IsIdempotent(proj[2]);
true
gap> RankOfTransformation(proj[3]);
2
gap> S := ImageSetOfTransformation(proj[2]);;
gap> IsIsomorphicDigraph(CycleDigraph(4), InducedSubdigraph(D, S));
true
\end{verbatim}

3.3.33 \texttt{DigraphDirectProductProjections}

\begin{verbatim}
\texttt{DigraphDirectProductProjections(digraph)} \hspace{1cm} (attribute)
\textbf{Returns:} A list of transformations.

If \textit{digraph} is a direct product digraph, \textit{digraph} = \texttt{DigraphDirectProduct(gr\_1, gr\_2, \ldots )}, then \texttt{DigraphDirectProductProjections} returns a list \texttt{proj} such that \texttt{proj[i]} is the projection onto the \texttt{i}-th coordinate of the product.

A projection is an idempotent endomorphism of \textit{digraph}. If \texttt{gr\_1}, \texttt{gr\_2}, \ldots are all loopless digraphs, then the image of \textit{digraph} under \texttt{proj[i]} is isomorphic to \texttt{gr\_i}.

Currently this attribute is only set upon creating an immutable digraph via \texttt{DigraphDirectProduct} and there is no way of calculating it for other digraphs.

For more information, see \texttt{DigraphDirectProduct (3.3.31)}

\begin{verbatim}
gap> D := DigraphDirectProduct(ChainDigraph(3), CycleDigraph(4), Digraph([ [2], [2] ]));;
true
gap> HasDigraphDirectProductProjections(D);
true
gap> proj := DigraphDirectProductProjections(D);
Length(proj);
3
gap> IsIdempotent(proj[2]);
true
gap> RankOfTransformation(proj[3]);
2
gap> P := DigraphRemoveAllMultipleEdges( ReducedDigraph(OnDigraphs(D, proj[2])) );
true
gap> IsIsomorphicDigraph(CycleDigraph(4), P);
true
\end{verbatim}
### 3.3.34 LineDigraph

- **LineDigraph(digraph)**  
  - **Return:** 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`.

  Note that the returned digraph is always a new immutable digraph, and the argument `digraph` is never modified.

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

### 3.3.35 LineUndirectedDigraph

- **LineUndirectedDigraph(digraph)**  
  - **Return:** 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.

  Note that the returned digraph is always a new immutable digraph, and the argument `digraph` is never modified.

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

### 3.3.36 DoubleDigraph

- **DoubleDigraph(digraph)**  
  - **Return:** 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 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`, together with the original edges and their duplicates.

  If `digraph` is mutable, then `digraph` is modified in-place. If `digraph` is immutable, then a new immutable digraph constructed as described above is returned.

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

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

\textbf{Returns:} A digraph.

Let \textit{digraph} be a digraph with vertex set \( V \). This function returns the bipartite double digraph of \textit{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 \textit{digraph}. The resulting graph is bipartite, since the original edges are not included in the resulting digraph.

If \textit{digraph} is mutable, then \textit{digraph} is modified in-place. If \textit{digraph} is immutable, then a new immutable digraph constructed as described above is returned.

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

3.3.38 DigraphAddAllLoops

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

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

\textbf{Returns:} A digraph.

For a digraph \textit{digraph} this operation returns a new digraph constructed from \textit{digraph}, such that a loop is added for every vertex which did not have a loop in \textit{digraph}.

If \textit{digraph} is immutable, then a new immutable digraph is returned. Otherwise, the loops are added to the loopless vertices of the mutable digraph \textit{digraph} in-place.

\begin{verbatim}
gap> D := EmptyDigraph(13);
<immutable empty digraph with 13 vertices>
gap> D := DigraphAddAllLoops(D);
<immutable reflexive digraph with 13 vertices, 13 edges>
gap> OutNeighbours(D);
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ], [ 6 ], [ 7 ], [ 8 ], [ 9 ],
  [ 10 ], [ 11 ], [ 12 ], [ 13 ] ]
gap> D := Digraph([[1, 2, 3], [1, 3], [1]]);
<immutable digraph with 3 vertices, 6 edges>
gap> D := DigraphAddAllLoops(D);
<immutable reflexive digraph with 3 vertices, 8 edges>
gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
gap> DigraphAddAllLoops(D);
<mutable digraph with 3 vertices, 6 edges>
gap> D;
<mutable digraph with 3 vertices, 6 edges>
\end{verbatim}

3.3.39 DistanceDigraph (for digraph and int)

\[ \text{DistanceDigraph}(\text{digraph}, i) \]

\[ \text{DistanceDigraph}(\text{digraph}, \text{list}) \]

\textbf{Operation:}
Returns: A digraph.

The first argument is a digraph, the second argument is a non-negative integer or a list of positive integers. This operation returns a digraph on the same set of vertices as `digraph`, with two vertices being adjacent if and only if the distance between them in `digraph` equals `i` or is a number in `list`. See `DigraphShortestDistance (5.3.2)`. If `digraph` is mutable, then `digraph` is modified in-place. If `digraph` is immutable, then a new immutable digraph constructed as described above is returned.

Example

```gap
digraph := DigraphFromSparse6String(">.Ln?AL'BC_DEbEF'GIaGHdIJeGKcKL_@McDHfILaBJfHMjKM");
<immutable digraph with 30 vertices, 90 edges>
digraph := DigraphShortestDistance(digraph, 1);
<immutable digraph with 30 vertices, 90 edges>
digraph := DigraphShortestDistance(digraph, 1, 2); <immutable digraph with 30 vertices, 270 edges>
```

3.3.40 DigraphClosure

▷ 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 (6.1.12), DigraphHasLoops (6.1.1), IsMultiDigraph (6.1.10), and OutDegreeOfVertex (5.2.10).` The operation `DigraphClosure` returns the k-closure of `digraph`.

Example

```gap
D := CompleteDigraph(6);
<immutable complete digraph with 6 vertices>
D := DigraphRemoveEdges(D, [[1, 2], [2, 1]]);
<immutable digraph with 6 vertices, 28 edges>
closure := DigraphClosure(D, 6);
<immutable digraph with 6 vertices, 30 edges>
IsCompleteDigraph(closure); true
```

3.3.41 DigraphMycielskian

▷ DigraphMycielskian(digraph) (operation)

▷ DigraphMycielskianAttr(digraph) (attribute)

Returns: A digraph.

If `digraph` is a symmetric digraph, then `DigraphMycielskian` returns the Mycielskian of `digraph`. The Mycielskian of a symmetric digraph is a larger symmetric digraph constructed from it, which has a larger chromatic number. For further information, see https://en.wikipedia.org/wiki/Mycielskian. If `digraph` is immutable, then a new immutable digraph is returned. Otherwise, the mutable digraph `digraph` is changed in-place into its Mycielskian.
Example

```gap
D := CycleDigraph(2);
gap> ChromaticNumber(D);
2
D := DigraphMycielskian(D);
gap> ChromaticNumber(D);
3
D := DigraphMycielskian(D);
gap> ChromaticNumber(D);
4
D := CompleteBipartiteDigraph(IsMutable, 2, 3);
gap> DigraphMycielskian(D);
gap> D;
```

### 3.4 Random digraphs

#### 3.4.1 RandomDigraph

```gap
RandomDigraph([filt, n[, p]])
```

**Returns:** A digraph.

If the optional first argument `filt` is present, then this should specify the category or representation the digraph being created will belong to. For example, if `filt` is `IsMutableDigraph` (3.1.2), then the digraph being created will be mutable, if `filt` is `IsImmutableDigraph` (3.1.3), then the digraph will be immutable. If the optional first argument `filt` is not present, then `IsImmutableDigraph` (3.1.3) is used by default.

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).

```gap
RandomDigraph(1000);
<immutable digraph with 1000 vertices, 364444 edges>
gap> RandomDigraph(10000, 0.023);
<immutable digraph with 10000 vertices, 2300438 edges>
gap> RandomDigraph(IsMutableDigraph, 1000, 1 / 2);
<mutable digraph with 1000 vertices, 499739 edges>
```

#### 3.4.2 RandomMultiDigraph

```gap
RandomMultiDigraph(n[, m])
```

**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.

\[
\text{Example}
\]
\[
gap> \text{RandomMultiDigraph}(1000);
\langle\text{immutable multidigraph with 1000 vertices, 216659 edges}\rangle
\]
\[
gap> \text{RandomMultiDigraph}(1000, 950);
\langle\text{immutable multidigraph with 1000 vertices, 950 edges}\rangle
\]

3.4.3 RandomTournament

\( \triangleright \text{RandomTournament(\{filt, \}n)} \)  

(operation)

Returns: A digraph.

If the optional first argument \( \text{filt} \) is present, then this should specify the category or representation the digraph being created will belong to. For example, if \( \text{filt} \) is \text{IsMutableDigraph} (3.1.2), then the digraph being created will be mutable, if \( \text{filt} \) is \text{IsImmutableDigraph} (3.1.3), then the digraph will be immutable. If the optional first argument \( \text{filt} \) is not present, then \text{IsImmutableDigraph} (3.1.3) is used by default.

If \( n \) is a non-negative integer, this function returns a random tournament with \( n \) vertices. See \text{IsTournament} (6.1.13).

\[
\text{Example}
\]
\[
\text{gap> RandomTournament(10);}
\langle\text{immutable tournament with 10 vertices}\rangle
\]
\[
\text{gap> RandomTournament(\text{IsMutableDigraph}, 10);}
\langle\text{mutable digraph with 1000 vertices, 500601 edges}\rangle
\]

3.4.4 RandomLattice

\( \triangleright \text{RandomLattice(\{n\)} \)

(operation)

Returns: A digraph.

If the optional first argument \( \text{filt} \) is present, then this should specify the category or representation the digraph being created will belong to. For example, if \( \text{filt} \) is \text{IsMutableDigraph} (3.1.2), then the digraph being created will be mutable, if \( \text{filt} \) is \text{IsImmutableDigraph} (3.1.3), then the digraph will be immutable. If the optional first argument \( \text{filt} \) is not present, then \text{IsImmutableDigraph} (3.1.3) is used by default.

If \( n \) is a positive integer, this function return a random lattice with \( m \) vertices, where it is guaranteed that \( m \) is between \( n \) and \( 2 \times n \). See \text{IsLatticeDigraph} (6.1.17).

\[
\text{Example}
\]
\[
\text{gap> RandomLattice(10);}
\langle\text{immutable lattice digraph with 10 vertices, 39 edges}\rangle
\]
\[
\text{gap> RandomLattice(\text{IsMutableDigraph}, 10);}
\langle\text{mutable digraph with 12 vertices, 52 edges}\rangle
\]
3.5 Standard examples

3.5.1 ChainDigraph

▷ ChainDigraph([filt, n])

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 \( i + 1 \).

If the optional first argument \( filt \) is present, then this should specify the category or representation the digraph being created will belong to. For example, if \( filt \) is IsMutableDigraph (3.1.2), then the digraph being created will be mutable, if \( filt \) is IsImmutableDigraph (3.1.3), then the digraph will be immutable. If the optional first argument \( filt \) is not present, then IsImmutableDigraph (3.1.3) is used by default.

The DigraphReflexiveTransitiveClosure (3.3.12) of a chain represents a total order.

Example.

```gap
gap> ChainDigraph(42);
<immutable chain digraph with 42 vertices>
gap> ChainDigraph(IsMutableDigraph, 10);
<mutable digraph with 10 vertices, 9 edges>
```

3.5.2 CompleteDigraph

▷ CompleteDigraph([filt, n])

Returns: A digraph.

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

If the optional first argument \( filt \) is present, then this should specify the category or representation the digraph being created will belong to. For example, if \( filt \) is IsMutableDigraph (3.1.2), then the digraph being created will be mutable, if \( filt \) is IsImmutableDigraph (3.1.3), then the digraph will be immutable. If the optional first argument \( filt \) is not present, then IsImmutableDigraph (3.1.3) is used by default.

Example.

```gap
gap> CompleteDigraph(20);
<immutable complete digraph with 20 vertices>
gap> CompleteDigraph(IsMutableDigraph, 10);
<mutable digraph with 10 vertices, 90 edges>
```

3.5.3 CompleteBipartiteDigraph

▷ CompleteBipartiteDigraph([filt, m, n])

Returns: A digraph.

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

If \( m \) and \( n \) are positive integers, this function returns the complete bipartite digraph with vertex sets of sizes \( m \) (containing the vertices \([1 \ldots m]\)) and \( n \) (containing the vertices \([m + 1 \ldots m + n]\)).
If the optional first argument \texttt{filt} is present, then this should specify the category or representation the digraph being created will belong to. For example, if \texttt{filt} is \texttt{IsMutableDigraph (3.1.2)}, then the digraph being created will be mutable, if \texttt{filt} is \texttt{IsImmutableDigraph (3.1.3)}, then the digraph will be immutable. If the optional first argument \texttt{filt} is not present, then \texttt{IsImmutableDigraph (3.1.3)} is used by default.

\begin{example}
\begin{verbatim}
gap> CompleteBipartiteDigraph(2, 3);
<immutable complete bipartite digraph with bicomponent sizes 2 and 3>
gap> CompleteBipartiteDigraph(IsMutableDigraph, 3, 2);
<mutable digraph with 5 vertices, 12 edges>
\end{verbatim}
\end{example}

### 3.5.4 CompleteMultipartiteDigraph

\begin{verbatim}
\texttt{CompleteMultipartiteDigraph([filt, \orders])} (operation)
\end{verbatim}

\begin{example}
\begin{verbatim}
\texttt{CompleteMultipartiteDigraph([5, 4, 2]);}
<immutable complete multipartite digraph with 11 vertices, 76 edges>
\texttt{CompleteMultipartiteDigraph(IsMutableDigraph, [5, 4, 2]);}
<mutable digraph with 11 vertices, 76 edges>
\end{verbatim}
\end{example}

### 3.5.5 CycleDigraph

\begin{verbatim}
\texttt{CycleDigraph([filt, \n])} (operation)
\end{verbatim}

\begin{example}
\begin{verbatim}
\texttt{CycleDigraph(1);}
<immutable digraph with 1 vertex, 1 edge>
\texttt{CycleDigraph(123);}
<immutable cycle digraph with 123 vertices>
\texttt{CycleDigraph(IsMutableDigraph, 10);}
<mutable digraph with 10 vertices, 10 edges>
\end{verbatim}
\end{example}
3.5.6 EmptyDigraph

\[ \text{EmptyDigraph}(\text{filt, } n) \]

(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.

If the optional first argument \( \text{filt} \) is present, then this should specify the category or representation the digraph being created will belong to. For example, if \( \text{filt} \) is \text{IsMutableDigraph} (3.1.2), then the digraph being created will be mutable, if \( \text{filt} \) is \text{IsImmutableDigraph} (3.1.3), then the digraph will be immutable. If the optional first argument \( \text{filt} \) is not present, then \text{IsImmutableDigraph} (3.1.3) is used by default.

\[ \text{NullDigraph}(\text{filt, } n) \]

(operation)

NullDigraph is a synonym for EmptyDigraph.

**Example**

\[ \text{EmptyDigraph}(20); \]
\(<\text{immutable empty digraph with 20 vertices}>\)
\[ \text{NullDigraph}(10); \]
\(<\text{immutable empty digraph with 10 vertices}>\)
\[ \text{EmptyDigraph}(<\text{IsMutableDigraph}, 10); \]
\(<\text{mutable empty digraph with 10 vertices}>\)

3.5.7 JohnsonDigraph

\[ \text{JohnsonDigraph}(\text{filt, } n, k) \]

(operation)

**Returns:** A digraph.

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

If the optional first argument \( \text{filt} \) is present, then this should specify the category or representation the digraph being created will belong to. For example, if \( \text{filt} \) is \text{IsMutableDigraph} (3.1.2), then the digraph being created will be mutable, if \( \text{filt} \) is \text{IsImmutableDigraph} (3.1.3), then the digraph will be immutable. If the optional first argument \( \text{filt} \) is not present, then \text{IsImmutableDigraph} (3.1.3) is used by default.

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

**Example**

\[ \text{gr := JohnsonDigraph}(3, 1); \]
\(<\text{immutable symmetric digraph with 3 vertices, 6 edges}>\)
\[ \text{OutNeighbours(gr)}; \]
\([ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ] \)
\[ \text{gr := JohnsonDigraph}(4, 2); \]
\(<\text{immutable symmetric digraph with 6 vertices, 24 edges}>\)
\[ \text{OutNeighbours(gr)}; \]
\([ [ 2, 3, 4, 5 ], [ 1, 3, 4, 6 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ], [ 1, 3, 4, 6 ], [ 2, 3, 4, 5 ] ] \)
\[ \text{JohnsonDigraph}(1, 0); \]
\(<\text{immutable empty digraph with 1 vertex}>\)
\[ \text{JohnsonDigraph}(\text{IsMutableDigraph}, 1, 0); \]
\(<\text{mutable empty digraph with 1 vertex}>\)
3.5.8 PetersenGraph

▷ PetersenGraph([filt])

Returns: A digraph.

From https://en.wikipedia.org/wiki/Petersen_graph:

“The Petersen graph is an undirected graph with 10 vertices and 15 edges. It is a small graph that serves as a useful example and counterexample for many problems in graph theory. The Petersen graph is named after Julius Petersen, who in 1898 constructed it to be the smallest bridgeless cubic graph with no three-edge-coloring.”

If the optional first argument filt is present, then this should specify the category or representation the digraph being created will belong to. For example, if filt is IsMutableDigraph (3.1.2), then the digraph being created will be mutable, if filt is IsImmutableDigraph (3.1.3), then the digraph will be immutable. If the optional first argument filt is not present, then IsImmutableDigraph (3.1.3) is used by default.

See also GeneralisedPetersenGraph (3.5.9).

Example

```
gap> ChromaticNumber(PetersenGraph());
3

gap> PetersenGraph(IsMutableDigraph);
<mutable digraph with 10 vertices, 30 edges>
```

3.5.9 GeneralisedPetersenGraph

▷ GeneralisedPetersenGraph([filt, ]n, k)

Returns: A digraph.

If n is a positive integer and k is a non-negative integer less than n / 2, then this operation returns the generalised Petersen graph GPG(n,k).

From https://en.wikipedia.org/wiki/Generalized_Petersen_graph:

“The generalized Petersen graphs are a family of cubic graphs formed by connecting the vertices of a regular polygon to the corresponding vertices of a star polygon. They include the Petersen graph and generalize one of the ways of constructing the Petersen graph. The generalized Petersen graph family was introduced in 1950 by H. S. M. Coxeter and was given its name in 1969 by Mark Watkins.”

If the optional first argument filt is present, then this should specify the category or representation the digraph being created will belong to. For example, if filt is IsMutableDigraph (3.1.2), then the digraph being created will be mutable, if filt is IsImmutableDigraph (3.1.3), then the digraph will be immutable. If the optional first argument filt is not present, then IsImmutableDigraph (3.1.3) is used by default.

See also PetersenGraph (3.5.8).

Example

```
gap> GeneralisedPetersenGraph(7, 2);
<immutable symmetric digraph with 14 vertices, 42 edges>

gap> GeneralisedPetersenGraph(40, 1);
<immutable symmetric digraph with 80 vertices, 240 edges>

gap> D := GeneralisedPetersenGraph(5, 2);
<immutable symmetric digraph with 10 vertices, 30 edges>

gap> IsIsomorphicDigraph(D, PetersenGraph());
true

gap> GeneralisedPetersenGraph(IsMutableDigraph, 9, 4);
<mutable digraph with 18 vertices, 54 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) (operation)
  
  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

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

51
Digraphs

<immutable digraph with 3 vertices, 3 edges>
 gap> IsSubdigraph(g, h);
 true
 gap> IsSubdigraph(h, g);
 false
 gap> IsSubdigraph(CompleteDigraph(4), CycleDigraph(4));
 true
 gap> IsSubdigraph(CycleDigraph(4), ChainDigraph(4));
 true
 gap> g := Digraph([ [2, 2], [1] ]);<immutable multidigraph with 2 vertices, 3 edges>
 gap> h := Digraph([ [2], [1] ]);<immutable digraph with 2 vertices, 2 edges>
 gap> IsSubdigraph(g, h);
 true
 gap> IsSubdigraph(h, g);
 false

4.1.2 IsUndirectedSpanningTree

\begin{itemize}
\item IsUndirectedSpanningTree(super, sub) \hspace{1cm} \text{(operation)}
\item IsUndirectedSpanningForest(super, sub) \hspace{1cm} \text{(operation)}
\end{itemize}

\textbf{Returns:} true or false.

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.

An 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 (3.3.5) is not connected has no undirected spanning trees (see IsConnectedDigraph (6.3.3)).

An 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 (3.3.5) of super (see DigraphConnectedComponents (5.3.9)).

Note that an undirected spanning tree is an undirected spanning forest that is connected.

\begin{itemize}
\item D := CompleteDigraph(4);<immutable complete digraph with 4 vertices>
\item tree := Digraph([ [3], [4], [1, 4], [2, 3] ]);<immutable digraph with 4 vertices, 6 edges>
\item IsSubdigraph(D, tree) and IsUndirectedTree(tree);
 true
\item IsUndirectedSpanningTree(D, tree);
 true
\item forest := EmptyDigraph(4);<immutable empty digraph with 4 vertices>
\item IsSubdigraph(D, forest) and IsUndirectedForest(forest);
 true
\item IsUndirectedSpanningForest(D, forest);
\end{itemize}
false
gap> IsSubdigraph(tree, forest);
true
gap> D := DigraphDisjointUnion(CycleDigraph(2), CycleDigraph(2));
<immutable digraph with 4 vertices, 4 edges>
gap> IsUndirectedTree(D);
false
gap> IsUndirectedForest(D) and IsUndirectedSpanningForest(D, D);
true
Chapter 5

Attributes and operations

5.1 Vertices and edges

5.1.1 DigraphVertices

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

Returns: A list of integers.

Returns the vertices of the digraph \textit{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"]);
<immutable digraph with 3 vertices, 3 edges>
gap> DigraphVertices(gr);
[ 1 .. 3 ]
gap> gr := Digraph([1, 2, 3, 4, 5, 7],
>                   [1, 2, 4, 4],
>                   [2, 7, 5, 3, 7]);
<immutable digraph with 6 vertices, 5 edges>
gap> DigraphVertices(gr);
[ 1 .. 6 ]
gap> DigraphVertices(RandomDigraph(100));
[ 1 .. 100 ]
gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
gap> DigraphVertices(D);
[ 1 .. 3 ]
\end{verbatim}

5.1.2 DigraphNrVertices

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

Returns: An integer.

Returns the number of vertices of the digraph \textit{digraph}.

Example

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

5.1.3 DigraphEdges

\[ \text{DigraphEdges}\] (attribute)

**Returns:** A list of lists.

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

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

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

**Example**

\[
\text{gap> gr := DigraphFromDiSparse6String(".DaXbOe?EAM@G~");}
\]
\[<\text{immutable multidigraph with 5 vertices, 16 edges}>\]
\[
\text{gap> edges := ShallowCopy(DigraphEdges(gr));; Sort(edges);}\]
\[\text{edges};\]
\[
[ [ 1, 1 ], [ 1, 3 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 1 ], [ 2, 2 ], [ 2, 3 ], [ 2, 5 ], [ 3, 2 ], [ 3, 4 ], [ 3, 5 ], [ 4, 2 ], [ 4, 4 ], [ 4, 5 ], [ 5, 1 ] ]\]
\[
\text{gap> D := CycleDigraph(IsMutableDigraph, 3);}\]
\[<\text{mutable digraph with 3 vertices, 3 edges}>\]
\[
\text{gap> DigraphEdges(D);}\]
\[ [ [ 1, 2 ], [ 2, 3 ], [ 3, 1 ] ] \]

5.1.4 DigraphNrEdges

\[ \text{DigraphNrEdges}\] (attribute)

**Returns:** An integer.

This function returns the number of edges of the digraph \( \text{digraph} \).

**Example**

\[
\text{gap> gr := Digraph([} \]
\[ > [1, 3, 4, 5], [1, 2, 3, 5], [2, 4, 5], [2, 4, 5], [1]]);\]
\[
\text{gap> DigraphNrEdges(gr);}\]
Digraphs

15
gap> gr := Digraph(["a", "b", "c"],
>                  ["a", "b", "b"],
>                  ["b", "a", "a"]);
<immutable multidigraph with 3 vertices, 3 edges>
gap> DigraphNrEdges(gr);
3
gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
gap> DigraphNrEdges(D);
3

5.1.5 DigraphSinks

\textbf{DigraphSinks}(\textit{digraph}) \hspace{1cm} \text{(attribute)}

\textbf{Returns:} A list of vertices.

This function returns a list of the sinks of the digraph \textit{digraph}. A sink of a digraph is a vertex with out-degree zero. See \text{OutDegreeOfVertex} (5.2.10).

Example

\begin{verbatim}
gap> gr := Digraph([[3, 5, 2, 2], [3], [], [5, 2, 5, 3], []]);
<immutable multidigraph with 5 vertices, 9 edges>
gap> DigraphSinks(gr);
[ 3, 5 ]
gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
gap> DigraphSinks(D);
[ ]
\end{verbatim}

5.1.6 DigraphSources

\textbf{DigraphSources}(\textit{digraph}) \hspace{1cm} \text{(attribute)}

\textbf{Returns:} A list of vertices.

This function returns an immutable list of the sources of the digraph \textit{digraph}. A source of a digraph is a vertex with in-degree zero. See \text{InDegreeOfVertex} (5.2.12).

Example

\begin{verbatim}
gap> gr := Digraph([[3, 5, 2, 2], [3], [], [5, 2, 5, 3], []]);
<immutable multidigraph with 5 vertices, 9 edges>
gap> DigraphSources(gr);
[ 1, 4 ]
gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
gap> DigraphSources(D);
[ ]
\end{verbatim}

5.1.7 DigraphTopologicalSort

\textbf{DigraphTopologicalSort}(\textit{digraph}) \hspace{1cm} \text{(attribute)}

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

If \textit{digraph} is a digraph whose only directed cycles are loops, then \text{DigraphTopologicalSort} returns the vertices of \textit{digraph} ordered so that every edge’s source appears no earlier in the list than
its range. If the digraph \textit{digraph} contains directed cycles of length greater than 1, then this operation returns \texttt{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.

\begin{verbatim}
gap> D := Digraph([ [2, 3], [], [4, 6], [5], [], [7, 8, 9], [], [], []]);
<immutable digraph with 9 vertices, 8 edges>
gap> DigraphTopologicalSort(D);
[ 2, 5, 4, 7, 8, 9, 6, 3, 1 ]
gap> D := Digraph(IsMutableDigraph, [[2, 3], [3], [4], []]);
<mutable digraph with 4 vertices, 4 edges>
gap> DigraphTopologicalSort(D);
[ 4, 3, 2, 1 ]
\end{verbatim}

### 5.1.8 DigraphVertexLabel

\begin{verbatim}
\texttt{DigraphVertexLabel(digraph, i)} (operation)
\texttt{SetDigraphVertexLabel(digraph, i, obj)} (operation)
\end{verbatim}

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

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

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

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

\begin{verbatim}
gap> D := DigraphFromDigraph6String("&DHUEe_");
<immutable digraph with 5 vertices, 11 edges>
gap> DigraphVertexLabel(D, 3);
3

gap> D := Digraph( ["a", "b", "c"], [], []);
<immutable empty digraph with 3 vertices>
gap> DigraphVertexLabel(D, 2);
"b"
gap> SetDigraphVertexLabel(D, 2, "d");
gap> DigraphVertexLabel(D, 2);
"d"
gap> D := InducedSubdigraph(D, [1, 2]);
<immutable empty digraph with 2 vertices>
gap> DigraphVertexLabel(D, 2);
"d"
gap> D := Digraph(IsMutableDigraph, ["e", "f", "g"], [], []);
<mutable empty digraph with 3 vertices>
gap> DigraphVertexLabel(D, 1);
"e"
gap> SetDigraphVertexLabel(D, 1, "h");
\end{verbatim}
gap> DigraphVertexLabel(D, 1);
"h"
gap> InducedSubdigraph(D, [1, 2]);
<mutable empty digraph with 2 vertices>
gap> DigraphVertexLabel(D, 1);
"h"

5.1.9 DigraphVertexLabels

- DigraphVertexLabels(digraph) (operation)
- SetDigraphVertexLabels(digraph, list) (operation)

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

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

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

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

Example

```gap
gap> D := DigraphFromDigraph6String("&DHUEe_"));
<immutable digraph with 5 vertices, 11 edges>
gap> DigraphVertexLabels(D);
[ 1 .. 5 ]
gap> D := Digraph(["a", "b", "c"], [], []);
<immutable empty digraph with 3 vertices>
gap> DigraphVertexLabels(D);
[ "a", "b", "c" ]
gap> SetDigraphVertexLabel(D, 2, "d");
gap> DigraphVertexLabels(D);
[ "a", "d", "c" ]
gap> D := InducedSubdigraph(D, [1, 3]);
<immutable empty digraph with 2 vertices>
```

```gap
gap> D := Digraph(IsMutableDigraph, ["e", "f", "g"], [], []);
<mutable empty digraph with 3 vertices>
```

```gap
gap> SetDigraphVertexLabels(D, ["h", "i", "j"]);
gap> DigraphVertexLabels(D);
[ "h", "i", "j" ]
gap> InducedSubdigraph(D, [1, 3]);
<mutable empty digraph with 2 vertices>
```

```gap
gap> DigraphVertexLabels(D);
[ "h", "j" ]
```
5.1.10 DigraphEdgeLabel

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

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
gap> D := DigraphFromDigraph6String("&DHUEe_");   
<immutable digraph with 5 vertices, 11 edges>
gap> DigraphEdgeLabel(D, 3, 1);                  
1
gap> SetDigraphEdgeLabel(D, 2, 5, [42]);         
gap> DigraphEdgeLabel(D, 2, 5);                  
[ 42 ]
gap> D := InducedSubdigraph(D, [2, 5]);          
<immutable digraph with 2 vertices, 3 edges>
gap> DigraphEdgeLabel(D, 1, 2);                  
[ 42 ]
gap> D := ChainDigraph(IsMutableDigraph, 5);     
<mutable digraph with 5 vertices, 4 edges>
gap> DigraphEdgeLabel(D, 2, 3);                  
1
gap> SetDigraphEdgeLabel(D, 4, 5, [1729]);       
gap> DigraphEdgeLabel(D, 4, 5);                  
[ 1729 ]
gap> InducedSubdigraph(D, [4, 5]);               
<mutable digraph with 2 vertices, 1 edge>
gap> DigraphEdgeLabel(D, 1, 2);                  
[ 1729 ]
```

5.1.11 DigraphEdgeLabels

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

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

5.1.12 DigraphInEdges

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

5.1.13 DigraphOutEdges

```
> D := Digraph([[2, 2], [3, 3], [4, 4], [1, 1]]);
<immutable multidigraph with 4 vertices, 8 edges>
```
5.1.14 **IsDigraphEdge** (for digraph and list)

- **IsDigraphEdge**(*digraph*, *list*)
  - **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.

- **IsDigraphEdge**(*digraph*, *u*, *v*)
  - **Operation**
  
  The second form simply returns true if [*u*, *v*] is an edge in *digraph*, and false otherwise.

Example

```gap
gap> D := Digraph([[2, 2], [6], [], [3], [], [1]]);
<immutable multidigraph with 6 vertices, 5 edges>
gap> IsDigraphEdge(D, [1, 1]);
false
gap> IsDigraphEdge(D, [1, 2]);
true
gap> IsDigraphEdge(D, [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> D := Digraph([[2, 1], [2, 3, 4], [3, 5], [1]]);
<immutable digraph with 5 vertices, 8 edges>
gap> IsMatching(D, [[2, 1], [3, 2]]);
false
gap> edges := [[3, 2]];
<immutable digraph with 5 vertices, 8 edges>
gap> IsMatching(D, edges);
true
gap> IsMaximalMatching(D, edges);
false
gap> edges := [[5, 1], [3, 3]];
```
Digraphs

```gap
gap> IsMaximalMatching(D, edges);
true
gap> IsPerfectMatching(D, edges);
false
gap> edges := [[1, 2], [3, 3], [4, 5]];;
gap> IsPerfectMatching(D, edges);
true
```

5.2 Neighbours and degree

5.2.1 AdjacencyMatrix

▷ AdjacencyMatrix(digraph) (attribute)
▷ AdjacencyMatrixMutableCopy(digraph) (operation)

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

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

The function AdjacencyMatrix returns an immutable list of lists, whereas the function AdjacencyMatrixMutableCopy returns a copy of AdjacencyMatrix that is a mutable list of mutable lists.

---

```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]]);
<immutable 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, 1, 1, 0, 1, 1 ],
  [ 0, 0, 0, 1, 0, 1, 0, 1, 1, 1 ],
  [ 0, 0, 2, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 1, 0, 0, 0, 0, 0, 0, 1 ],
  [ 1, 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 1, 0, 1, 1, 1, 0, 0, 0, 0, 0 ],
  [ 1, 0, 1, 1, 1, 0, 0, 0, 0, 0 ],
  [ 0, 1, 1, 1, 1, 0, 0, 0, 0, 0 ] ]
gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
gap> Display(AdjacencyMatrix(D));
[ [ 0, 1, 0 ],
  [ 0, 0, 1 ],
  [ 1, 0, 0 ] ]
```
5.2.2 CharacteristicPolynomial

CharacteristicPolynomial(digraph) (attribute)

Returns: A polynomial with integer coefficients.

This function returns the characteristic polynomial of the digraph \textit{digraph}. That is, it returns the characteristic polynomial of the adjacency matrix of the digraph \textit{digraph}.

Example

\begin{verbatim}
gap> D := 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] ]); <immutable multidigraph with 10 vertices, 44 edges> gap> CharacteristicPolynomial(D); \[ x_1^{10} - 3 \times x_1^9 - 7 \times x_1^8 - x_1^7 + 14 \times x_1^6 + x_1^5 - 26 \times x_1^4 + 51 \times x_1^3 - 10 \times x_1^2 + 18 \times x_1 - 30 \] gap> D := CompleteDigraph(5); <immutable complete digraph with 5 vertices> gap> CharacteristicPolynomial(D); \[ x_1^5 - 10 \times x_1^3 - 20 \times x_1^2 - 15 \times x_1 - 4 \] gap> D := CycleDigraph(IsMutableDigraph, 3); <mutable digraph with 3 vertices, 3 edges> gap> CharacteristicPolynomial(D); \[ x_1^3 - 1 \]
\end{verbatim}

5.2.3 BooleanAdjacencyMatrix

BooleanAdjacencyMatrix(digraph) (attribute)

BooleanAdjacencyMatrixMutableCopy(digraph) (operation)

Returns: A square matrix of booleans.

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

\begin{verbatim}
gap> gr := Digraph([[3, 4], [2, 3], [1, 2, 4], [4]]); <immutable digraph with 4 vertices, 8 edges> gap> PrintArray(BooleanAdjacencyMatrix(gr)); \[
[ \text{false, false, true, true} ],
[ \text{false, true, true, false} ],
[ \text{true, true, false, true} ],
[ \text{false, false, false, true} ]
\]
gap> gr := CycleDigraph(4);;

gap> PrintArray(BooleanAdjacencyMatrix(gr));\[
[ \text{false, true, false, false} ],
[ \text{false, false, true, false} ],
[ \text{false, false, false, true} ],
[ \text{false, true, false, false} ],
\]
\end{verbatim}
Digraphs

DigraphAdjacencyFunction

\( \triangleright \) DigraphAdjacencyFunction(digraph) (attribute)

**Returns:** A function.

If \( \text{digraph} \) is a digraph, then \( \text{DigraphAdjacencyFunction} \) returns a function which takes two integer parameters \( x, y \) and returns \( \text{true} \) if there exists an edge from vertex \( x \) to vertex \( y \) in \( \text{digraph} \) and \( \text{false} \) if not.

\[
\begin{align*}
gap> \text{digraph} := \text{Digraph}([[1, 2], [3], []]); \\
<\text{immutable digraph with 3 vertices, 3 edges}> \\
gap> \text{foo} := \text{DigraphAdjacencyFunction}(\text{digraph}); \\
defunction( u, v ) \ldots \text{end} \\
gap> \text{foo}(1, 1); \\
\text{true} \\
gap> \text{foo}(1, 2); \\
\text{true} \\
gap> \text{foo}(1, 3); \\
\text{false} \\
gap> \text{foo}(3, 1); \\
\text{false} \\
gap> \text{gr} := \text{Digraph}(["a", "b", "c"], \\
> ["a", "b", "b"], \\
> ["b", "a", "a"]); \\
<\text{immutable multidigraph with 3 vertices, 3 edges}> \\
gap> \text{foo} := \text{DigraphAdjacencyFunction}(\text{gr}); \\
defunction( u, v ) \ldots \text{end} \\
gap> \text{foo}(1, 2); \\
\text{true} \\
gap> \text{foo}(3, 2); \\
\text{false} \\
gap> \text{foo}(3, 1); \\
\text{false} \\
gap> \text{D} := \text{CycleDigraph}(\text{IsMutableDigraph}, 3); \\
<\text{mutable digraph with 3 vertices, 3 edges}> \\
gap> \text{foo} := \text{DigraphAdjacencyFunction}(\text{D}); \\
defunction( u, v ) \ldots \text{end} \\
gap> \text{foo}(1, 2); \\
\text{true} \\
gap> \text{foo}(2, 1); \\
\text{false}
\]
5.2.5 DigraphRange

\[\text{DigraphRange}(\text{digraph})\] (attribute)  
\[\text{DigraphSource}(\text{digraph})\] (attribute)  

**Returns:** A list of positive integers. 

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

**Example**

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

5.2.6 OutNeighbours

\[\text{OutNeighbours}(\text{digraph})\] (attribute)  
\[\text{OutNeighbors}(\text{digraph})\] (attribute)  
\[\text{OutNeighboursMutableCopy}(\text{digraph})\] (operation)  
\[\text{OutNeighborsMutableCopy}(\text{digraph})\] (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 lists, whereas the function `OutNeighboursMutableCopy` returns a copy of `OutNeighbours` which is a mutable list of mutable lists.

**Example**

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

gap> gr := DigraphByAdjacencyMatrix([ [1, 2, 1], [1, 1, 0], [0, 0, 1]));
<immutable multidigraph with 3 vertices, 7 edges>

gap> OutNeighbours(gr);
[ [ 1, 2, 2, 3 ], [ 1, 2 ], [ 3 ] ]

gap> OutNeighboursMutableCopy(gr);
[ [ 1, 2, 2, 3 ], [ 1, 2 ], [ 3 ] ]

gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>

gap> OutNeighbours(D);
[ [ 2 ], [ 3 ], [ 1 ] ]

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

gap> InNeighbours(gr);
[ [ 2 ], [ 1 ], [ 2 ] ]

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

gap> InNeighbours(gr);
[ [ 1, 2 ], [ 1, 2 ], [ 1, 3 ] ]

gap> gr := DigraphByAdjacencyMatrix([ [1, 2, 1], [1, 1, 0], [0, 0, 1]]);
<immutable multidigraph with 3 vertices, 7 edges>

gap> InNeighbours(gr);
[ [ 1, 2 ], [ 1, 1, 2 ], [ 1, 3 ] ]

gap> InNeighboursMutableCopy(gr);
[ [ 1, 2 ], [ 1, 1, 2 ], [ 1, 3 ] ]

gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
```
Digraphs

5.2.8 OutDegrees

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

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

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

Returns: A list of non-negative integers.

Given a digraph \textit{digraph} with \( n \) vertices, the function \texttt{OutDegrees} returns an immutable list \texttt{out} of length \( n \), such that for a vertex \( i \) in \textit{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.

Example

\begin{verbatim}
gap> D := Digraph([[1, 3, 2, 2], [], [2, 1], []]);
<immutable multidigraph with 4 vertices, 6 edges>
gap> OutDegrees(D);
[ 4, 0, 2, 0 ]
gap> OutDegreeSequence(D);
[ 4, 2, 0, 0 ]
gap> OutDegreeSet(D);
[ 0, 2, 4 ]
gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
gap> OutDegrees(D);
[ 1, 1, 1 ]
\end{verbatim}

5.2.9 InDegrees

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

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

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

Returns: A list of non-negative integers.

Given a digraph \textit{digraph} with \( n \) vertices, the function \texttt{InDegrees} returns an immutable list \texttt{inn} of length \( n \), such that for a vertex \( i \) in \textit{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.

Example

\begin{verbatim}
gap> D := Digraph([[1, 3, 2, 2, 4], [], [2, 1, 4], []]);
<immutable multidigraph with 4 vertices, 8 edges>
gap> InDegrees(D);
[ 2, 3, 1, 2 ]
\end{verbatim}
Digraphs

5.2.10 OutDegreeOfVertex

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

\[ \text{OutDegreeOfVertex}\text{Digraph} \quad \text{vertex} \]

**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 \( \text{vertex} \).

**Example**

\[ \text{gap> D := Digraph([} \]
\[ \quad \text{[2, 2, 1], [1, 4], [2, 2, 4, 2], [1, 1, 2, 2, 1, 2, 2]];} \]
\[ \text{<immutable multidigraph with 4 vertices, 16 edges>} \]
\[ \text{gap> OutDegreeOfVertex(D, 1);} \]
\[ 3 \]
\[ \text{gap> OutDegreeOfVertex(D, 2);} \]
\[ 2 \]
\[ \text{gap> OutDegreeOfVertex(D, 3);} \]
\[ 4 \]
\[ \text{gap> OutDegreeOfVertex(D, 4);} \]
\[ 7 \]

5.2.11 OutNeighboursOfVertex

\[ \text{OutNeighboursOfVertex} \quad \text{digraph, vertex} \]

**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.

**Example**

\[ \text{gap> D := Digraph([} \]
\[ \quad \text{[2, 2, 3], [1, 3, 4], [2, 2, 3], [1, 1, 2, 2, 1, 2, 2]];} \]
\[ \text{<immutable multidigraph with 4 vertices, 16 edges>} \]
\[ \text{gap> OutNeighboursOfVertex(D, 1);} \]
\[ [2, 2, 3] \]
\[ \text{gap> OutNeighboursOfVertex(D, 3);} \]
\[ [2, 2, 3] \]

5.2.12 InDegreeOfVertex

\[ \text{InDegreeOfVertex} \quad \text{digraph, vertex} \]

**Returns:** A non-negative integer.
This operation returns the in-degree of the vertex \textit{vertex} in the digraph \textit{digraph}. The in-degree of \textit{vertex} is the number of edges in \textit{digraph} whose range is \textit{vertex}.

\begin{verbatim}
gap> D := Digraph([ [2, 2, 1], [1, 4], [2, 2, 4, 2], [1, 1, 2, 2, 1, 2, 2] ]);;
<immutable multidigraph with 4 vertices, 16 edges>
gap> InDegreeOfVertex(D, 1); 5
gap> InDegreeOfVertex(D, 2); 9
gap> InDegreeOfVertex(D, 3); 0
gap> InDegreeOfVertex(D, 4); 2
\end{verbatim}

\subsection{InNeighboursOfVertex}

\begin{verbatim}
\textbf{InNeighboursOfVertex} (\texttt{digraph, vertex}) \hspace{1cm} (operation)
\textbf{InNeighborsOfVertex} (\texttt{digraph, vertex}) \hspace{1cm} (operation)

Returns: A list of positive vertices.

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

\begin{verbatim}
\gap> D := Digraph([ [2, 2, 3], [1, 3, 4], [2, 2, 3], [1, 1, 2, 2, 1, 2, 2] ]);;
<immutable multidigraph with 4 vertices, 16 edges>
\gap> InNeighboursOfVertex(D, 1); [ 2, 4, 4, 4 ]
\gap> InNeighboursOfVertex(D, 2); [ 1, 1, 3, 3, 4, 4, 4, 4 ]
\end{verbatim}

\subsection{DigraphLoops}

\begin{verbatim}
\textbf{DigraphLoops} (\texttt{digraph}) \hspace{1cm} (attribute)

Returns: A list of vertices.

If \textit{digraph} is a digraph, then \textbf{DigraphLoops} returns the list consisting of the \textbf{DigraphVertices} (5.1.1) of \textit{digraph} at which there is a loop. See \textbf{DigraphHasLoops} (6.1.1).

\begin{verbatim}
\gap> D := Digraph([ [3, 5], [1], [2, 4, 3], [4], [2, 1] ]);;
<immutable digraph with 5 vertices, 9 edges>
\gap> DigraphLoops(D); [ 3, 4 ]
\gap> D := Digraph(IsMutableDigraph, [[1], [1]]);
\end{verbatim}
\end{verbatim}
5.2.15 PartialOrderDigraphMeetOfVertices

\begin{itemize}
  \item \texttt{PartialOrderDigraphMeetOfVertices\hspace{1em} (operation)}
  \item \texttt{PartialOrderDigraphJoinOfVertices\hspace{1em} (operation)}
\end{itemize}

\textbf{Returns:} A positive integer or \texttt{fail}

If the first argument is a partial order digraph \texttt{IsPartialOrderDigraph (6.1.16)} then these operations return the meet, or the join, of the two input vertices. If the meet (or join) does not exist then \texttt{fail} is returned. The meet (or join) is guaranteed to exist when the first argument satisfies \texttt{IsMeetSemilatticeDigraph (6.1.17)} (or \texttt{IsJoinSemilatticeDigraph (6.1.17)}) - see the documentation for these properties for the definition of the meet (or the join).

\begin{verbatim}
gap> D := Digraph([[1, 2], [1, 3], [1, 2, 3, 4]]);
<immutable digraph with 4 vertices, 9 edges>
gap> PartialOrderDigraphMeetOfVertices(D, 2, 3);
4
gap> PartialOrderDigraphJoinOfVertices(D, 2, 3);
1
\end{verbatim}

5.2.16 DegreeMatrix

\begin{itemize}
  \item \texttt{DegreeMatrix\hspace{1em} (attribute)}
\end{itemize}

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

This function returns the outdegree matrix \texttt{mat} of the digraph \texttt{digraph}. The value of the \texttt{i}th diagonal matrix entry is the outdegree of the vertex \texttt{i} in \texttt{digraph}. All off-diagonal entries are 0. If \texttt{digraph} has no vertices, then the empty list is returned.

See \texttt{OutDegrees (5.2.8)} for more information.

\begin{verbatim}
gap> D := Digraph([[1, 2, 3], [4], [1, 3, 4], []]);
<immutable digraph with 4 vertices, 7 edges>
gap> PrintArray(DegreeMatrix(D));

\end{verbatim}
Digraphs

\[ \begin{bmatrix} 0,0,3,0 \\ 0,0,0,0 \end{bmatrix} \]

\textbf{gap> D := CycleDigraph(5);;}
\textbf{gap> PrintArray(DegreeMatrix(D));}
\[ \begin{bmatrix} 1,0,0,0,0 \\ 0,1,0,0,0 \\ 0,0,1,0,0 \\ 0,0,0,1,0 \\ 0,0,0,0,1 \end{bmatrix} \]

\textbf{gap> DegreeMatrix(EmptyDigraph(0));}
\[
\]

\textbf{gap> D := CycleDigraph(IsMutableDigraph, 3);<mutable digraph with 3 vertices, 3 edges>
\textbf{gap> Display(DegreeMatrix(D));}
\[ \begin{bmatrix} 1,0,0 \\ 0,1,0 \\ 0,0,1 \end{bmatrix} \]

\subsection*{5.2.17 LaplacianMatrix}

\textbf{LaplacianMatrix(digraph)} (attribute)

\textbf{Returns:} A square matrix of integers. This function returns the outdegree Laplacian matrix mat of the digraph \textit{digraph}. The outdegree Laplacian matrix is defined as \textbf{DegreeMatrix(digraph)} - \textbf{AdjacencyMatrix(digraph)}. If \textit{digraph} has no vertices, then the empty list is returned.

See \textbf{DegreeMatrix (5.2.16)} and \textbf{AdjacencyMatrix (5.2.1)} for more information.

\textbf{Example}

\textbf{gap> gr := Digraph([[1, 2, 3], [4], [1, 3, 4], []]);<immutable digraph with 4 vertices, 7 edges>
\textbf{gap> PrintArray(LaplacianMatrix(gr));}
\[ \begin{bmatrix} 2,-1,-1,0 \\ 0,1,0,-1 \\ -1,0,2,-1 \\ 0,0,0,0 \end{bmatrix} \]

\textbf{gap> LaplacianMatrix(EmptyDigraph(0));<default digraph>}
\[
\]

\textbf{gap> D := CycleDigraph(IsMutableDigraph, 3);<mutable digraph with 3 vertices, 3 edges>
\textbf{gap> Display(LaplacianMatrix(D));}
\[ \begin{bmatrix} 1,-1,0 \\ 0,1,-1 \\ -1,0,1 \end{bmatrix} \]

\subsection*{5.3 Reachability and connectivity}

\subsection*{5.3.1 DigraphDiameter}

\textbf{DigraphDiameter(digraph)} (attribute)

\textbf{Returns:} An integer or \textbf{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 \textit{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 \textbf{DigraphShortestDistances (5.3.3)}. 

\texttt{Example}

\begin{verbatim}
gap> D := Digraph([ [2], [3], [4, 5], [5], [1, 2, 3, 4, 5] ]);  
<immutable digraph with 5 vertices, 10 edges>  
gap> DigraphDiameter(D);  
3  
gap> D := ChainDigraph(2);  
<immutable chain digraph with 2 vertices>  
gap> DigraphDiameter(D);  
fail  
gap> IsStronglyConnectedDigraph(D);  
false  
gap> D := GeneralisedPetersenGraph(IsMutableDigraph, 6, 2);  
<mutable digraph with 12 vertices, 36 edges>  
gap> DigraphDiameter(D);  
4
\end{verbatim}

\subsection*{5.3.2 \textbf{DigraphShortestDistance (for a digraph and two vertices)}}

\begin{itemize}
\item \texttt{DigraphShortestDistance(digraph, u, v)} \hspace{2cm} (operation) 
\item \texttt{DigraphShortestDistance(digraph, list)} \hspace{2cm} (operation) 
\item \texttt{DigraphShortestDistance(digraph, list1, list2)} \hspace{2cm} (operation)
\end{itemize}

\textbf{Returns:} An integer or \texttt{fail}

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

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

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

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

\texttt{Example}

\begin{verbatim}
gap> D := Digraph([ [2], [3], [1, 4], [1, 3], [5] ]);  
<immutable digraph with 5 vertices, 7 edges>  
gap> DigraphShortestDistance(D, 1, 3);  
2  
gap> DigraphShortestDistance(D, [3, 3]);  
0  
gap> DigraphShortestDistance(D, 5, 2);  
fail  
gap> DigraphShortestDistance(D, [1, 2], [4, 5]);  
2  
gap> DigraphShortestDistance(D, [1, 3], [3, 5]);  
0
\end{verbatim}
5.3.3 DigraphShortestDistances

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

**Returns:** A square matrix.

If \textit{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
D := Digraph([[1, 2], [3], [1, 2], [4]]);
<immutable digraph with 4 vertices, 6 edges>
mat := DigraphShortestDistances(D);
PrintArray(mat);
[ [ 0, 1, 2, fail ],
  [ 2, 0, 1, fail ],
  [ 1, 1, 0, fail ],
  [ fail, fail, fail, 0 ] ]
```

5.3.4 DigraphLongestDistanceFromVertex

\[ \text{DigraphLongestDistanceFromVertex(digraph, v)} \]

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

If \textit{digraph} is a digraph and \( v \) is a vertex in \textit{digraph}, then this operation returns the length of the longest directed walk in \textit{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 \text{DigraphRemoveLoops} (3.3.24):
  \[ \text{DigraphLongestDistanceFromVertex(DigraphRemoveLoops(digraph, v))} \]
- 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 \textit{digraph}. See \text{DigraphSinks} (5.1.5).

**Example**

```gap
D := Digraph([[2], [3, 4], [], [5], [], [6]]);
<immutable digraph with 6 vertices, 5 edges>
DigraphLongestDistanceFromVertex(D, 1);
3
DigraphLongestDistanceFromVertex(D, 3);
```
5.3.5 DigraphDistanceSet (for a digraph, a pos int, and an int)

\[ \text{DigraphDistanceSet}(\text{digraph}, \text{vertex}, \text{distance}) \] (operation)

- \text{DigraphDistanceSet}(\text{digraph}, \text{vertex}, \text{distances}) (operation)

\textbf{Returns:} A list of vertices

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

- \text{digraph} should be a digraph,
- \text{vertex} should be a positive integer,
- \text{distance} should be a non-negative integer, and
- \text{distances} should be a list of non-negative integers.

\textbf{Example}

\begin{verbatim}
gap> D := Digraph([[2], [3], [1, 4], [1, 3]]); <immutable digraph with 4 vertices, 6 edges> gap> DigraphDistanceSet(D, 2, [1, 2]); [ 3, 1, 4 ] gap> DigraphDistanceSet(D, 3, 1); [ 1, 4 ] gap> DigraphDistanceSet(D, 2, 0); [ 2 ]
\end{verbatim}

5.3.6 DigraphGirth

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

\textbf{Returns:} An integer, or \textit{infinity}.

This attribute returns the \textit{girth} of the digraph \text{digraph}. The \textit{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 \text{digraph} has no directed cycles, then this function will return \textit{infinity}. If \text{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 \text{digraph}. If the digraph has known automorphisms [see DigraphGroup (7.2.10)], then the performance is likely to be better.

For symmetric digraphs, see also DigraphUndirectedGirth (5.3.8).

\textbf{Example}

\begin{verbatim}
gap> D := Digraph([[1], [1]]); <immutable digraph with 2 vertices, 2 edges> gap> DigraphGirth(D); 1 gap> D := Digraph([[2, 3], [3], [4], [1]]); <immutable digraph with 4 vertices, 4 edges> gap> DigraphGirth(D); infinity gap> D := Digraph([[2, 3], [3], [4], [1]]); <immutable digraph with 4 vertices, 5 edges> gap> DigraphGirth(D);
\end{verbatim}
5.3.7 DigraphOddGirth

\( \text{DigraphOddGirth}(\text{digraph}) \)

Returns: An integer, or infinity.

This attribute returns the odd girth of the digraph \text{digraph}. The odd girth of a digraph is the length of its shortest simple circuit of odd length. See Section 1.1.1 for the definitions of simple circuit, directed cycle, and loop.

If \text{digraph} has no directed cycles of odd length, then this function will return infinity, even if \text{digraph} has a directed cycle of even length. If \text{digraph} contains a loop, then this function will return 1.

See also DigraphGirth (5.3.6).

Example

\begin{verbatim}
gap> D := Digraph([[2, [3, 1], [1]]]);
<immutable digraph with 3 vertices, 4 edges>
gap> DigraphOddGirth(D);
3
gap> D := CycleDigraph(4);
<immutable cycle digraph with 4 vertices>
gap> DigraphOddGirth(D);
3
\end{verbatim}

5.3.8 DigraphUndirectedGirth

\( \text{DigraphUndirectedGirth}(\text{digraph}) \)

Returns: An integer or infinity.

If \text{digraph} is a symmetric digraph, then this function returns the girth of \text{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 \text{digraph} has no cycles, then this function will return infinity.
Example

```gap
D := Digraph([[2, 4], [1, 3], [2, 4], [1, 3]]);
<immutable digraph with 4 vertices, 8 edges>
D := Digraph([2, 4], [1, 3], [2, 4], [1, 3]);
<immutable digraph with 3 vertices, 4 edges>
D := GeneralisedPetersenGraph(IsMutableDigraph, 9, 2);
<mutable digraph with 18 vertices, 54 edges>
```

5.3.9 DigraphConnectedComponents

- **DigraphConnectedComponents(digraph)** (attribute)
- **DigraphNrConnectedComponents(digraph)** (attribute)

**Returns:** A record.

This function returns the record `wcc` corresponding to the weakly connected components of the digraph `digraph`. Two vertices of `digraph` are in the same weakly connected component whenever they are equal, or there exists a 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 (5.3.11)`)

of the `DigraphSymmetricClosure (3.3.11)` 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.

`DigraphNrConnectedComponents(digraph)` is simply a shortcut for `Length(DigraphConnectedComponents(digraph).comps)`, and is no more efficient.

Example

```gap
gr := Digraph([[2], [3, 1], [1]]);
<immutable digraph with 3 vertices, 3 edges>
gr := Digraph([[1], [1, 2], [3]]);
<immutable digraph with 3 vertices, 3 edges>
gr := EmptyDigraph(0);
<immutable empty digraph with 0 vertices>
```
gap> DigraphConnectedComponents(gr);
rec( comps := [ ], id := [ ] )
gap> D := CycleDigraph(IsMutableDigraph, 2);
<mutable digraph with 2 vertices, 2 edges>
gap> G := CycleDigraph(3);
<immutable cycle digraph with 3 vertices>
gap> DigraphDisjointUnion(D, G);
<mutable digraph with 5 vertices, 5 edges>
gap> DigraphConnectedComponents(D);
rec( comps := [ [ 1, 2 ], [ 3, 4, 5 ] ], id := [ 1, 1, 2, 2, 2 ] )

5.3.10 DigraphConnectedComponent

▷ DigraphConnectedComponent(digraph, vertex) (operation)

Returns: A list of vertices.

If vertex is a vertex in the digraph digraph, then this operation returns the connected component of vertex in digraph. See DigraphConnectedComponents (5.3.9) for more information.

Example

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

5.3.11 DigraphStronglyConnectedComponents

▷ DigraphStronglyConnectedComponents(digraph) (attribute)
▷ DigraphNrStronglyConnectedComponents(digraph) (attribute)

Returns: A record.

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

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

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

DigraphNrStronglyConnectedComponents(digraph) is simply a shortcut for Length(DigraphStronglyConnectedComponents(digraph).comps), and is no more efficient.

Example

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

rec( comps := [ [ 3 ], [ 1, 2 ] ], id := [ 2, 2, 1 ] )
gap> DigraphNrStronglyConnectedComponents(gr);
2
gap> D := DigraphDisjointUnion(CycleDigraph(4), CycleDigraph(5));
<immutable digraph with 9 vertices, 9 edges>
gap> DigraphStronglyConnectedComponents(D);
rec( comps := [ [ 1, 2, 3, 4 ], [ 5, 6, 7, 8, 9 ] ],
     id := [ 1, 1, 1, 1, 2, 2, 2, 2, 2 ] )
gap> DigraphNrStronglyConnectedComponents(D);
2
gap> D := CycleDigraph(IsMutableDigraph, 2);
<immutable digraph with 2 vertices, 2 edges>
gap> G := CycleDigraph(3);
<mutable cycle digraph with 3 vertices>
gap> DigraphDisjointUnion(D, G);
<mutable digraph with 5 vertices, 5 edges>
gap> DigraphStronglyConnectedComponents(D);
rec( comps := [ [ 1, 2 ], [ 3, 4, 5 ] ], id := [ 1, 1, 2, 2, 2 ] )

5.3.12 DigraphStronglyConnectedComponent

▷ DigraphStronglyConnectedComponent(digraph, vertex) (operation)

Returns: A list of vertices.

If vertex is a vertex in the digraph digraph, then this operation returns the strongly connected component of vertex in digraph. See DigraphStronglyConnectedComponents (5.3.11) for more information.

Example

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

5.3.13 DigraphBicomponents

▷ DigraphBicomponents(digraph) (attribute)

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

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

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

For a bipartite digraph with at least 3 vertices, there is a unique pair of bicomponents of bipartite if and only if the digraph is connected. See IsConnectedDigraph (6.3.3) for more information.
Example

```gap
D := CycleDigraph(3);
<immutable cycle digraph with 3 vertices>
D := DigraphBicomponents(D);
fail
D := ChainDigraph(5);
<immutable chain digraph with 5 vertices>
D := DigraphBicomponents(D);
[ [ 1, 3, 5 ], [ 2, 4 ] ]
D := Digraph([[5], [1, 4], [5], [5], []]);
<immutable digraph with 5 vertices, 5 edges>
D := DigraphBicomponents(D);
[ [ 1, 3, 4 ], [ 2, 5 ] ]
D := CompleteBipartiteDigraph(IsMutableDigraph, 2, 3);
<mutable digraph with 5 vertices, 12 edges>
D := DigraphBicomponents(D);
[ [ 1, 2 ], [ 3, 4, 5 ] ]
```

5.3.14 ArticulationPoints

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

\textbf{Returns:} A list of vertices.

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

ArticulationPoints returns a list of the articulation points of \textit{digraph}, if any, and, in particular, returns the empty list if \textit{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 \texttt{IsBiconnectedDigraph} (6.3.4).

Example

```gap
ArticulationPoints(CycleDigraph(5));
[ ]
D := Digraph([[2, 7], [3, 5], [4], [2], [6], [1], []]);
ArticulationPoints(D);
[ 2, 1 ]
ArticulationPoints(ChainDigraph(5));
[ 4, 3, 2 ]
ArticulationPoints(NullDigraph(5));
[ ]
D := ChainDigraph(IsMutableDigraph, 4);
<mutable digraph with 4 vertices, 3 edges>
ArticulationPoints(D);
[ 3, 2 ]
```
### 5.3.15 DigraphPeriod

**Function**

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

**Returns:** An integer.

This function returns the period of the digraph \text{digraph}.

If a digraph \text{digraph} has at least one directed cycle, then the period is the greatest positive integer which divides the lengths of all directed cycles of \text{digraph}. If \text{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 aperiodic. See \text{IsAperiodicDigraph} (6.3.6).

**Example**

```gap
gap> D := Digraph([[6], [1], [2], [3], [4, 4], [5]]);
<immutable multidigraph with 6 vertices, 7 edges>
gap> DigraphPeriod(D);
6
gap> D := Digraph([[2], [3, 5], [4], [5], [1, 2]]);
<immutable digraph with 5 vertices, 7 edges>
gap> DigraphPeriod(D);
1
gap> D := ChainDigraph(2);
<immutable chain digraph with 2 vertices>
gap> DigraphPeriod(D);
0
gap> IsAcyclicDigraph(D);
true
gap> D := GeneralisedPetersenGraph(IsMutableDigraph, 5, 2);
<mutable digraph with 10 vertices, 30 edges>
gap> DigraphPeriod(D);
1
```

### 5.3.16 DigraphFloydWarshall

**Function**

\[
\text{DigraphFloydWarshall}(\text{digraph}, \text{func}, \text{nopath}, \text{edge})
\]

**Returns:** A matrix.

If \text{digraph} is a digraph with \(n\) vertices, then this operation returns an \(n \times n\) matrix \text{mat} containing the output of a generalised version of the Floyd-Warshall algorithm, applied to \text{digraph}.

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

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

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

```gap
gap> D := DigraphFromDigraph6String("&EAHQeDB");
<immutable digraph with 6 vertices, 12 edges>

gap> func := function(mat, i, j, k)
> if mat[i][k] <> -1 and mat[k][j] <> -1 then
> if (mat[i][j] = -1) or (mat[i][j] > mat[i][k] + mat[k][j]) then
> mat[i][j] := mat[i][k] + mat[k][j];
> fi;
> fi;
> end;

gap> shortest_distances := DigraphFloydWarshall(D, 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, 2, 1, 1 ],
  [ 3, -1, -1, 2, 1, 1 ] ]
```

### 5.3.17 \texttt{IsReachable}

\texttt{IsReachable(digraph, u, v)}

\textbf{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 $\text{digraph}$, 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 \texttt{IsReachable} has worst case complexity of $O(m+n)$ where $m$ is the number of edges and $n$ the number of vertices in $\text{digraph}$.

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

gap> IsReachable(D, 1, 3);
true

gap> IsReachable(D, 2, 1);
false

gap> IsReachable(D, 3, 3);
true

gap> IsReachable(D, 1, 1);
false
```

### 5.3.18 \texttt{DigraphPath}

\texttt{DigraphPath(digraph, u, v)}

\textbf{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 \( \text{digraph} \), then this operation returns such a directed path (or directed cycle). Otherwise, this operation returns \( \text{fail} \). See Section ‘Definitions’ for the definition of a directed path and a directed cycle.

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

- \( v \) is the list \([v_1, v_2, \ldots, v_n]\).
- \( a \) is the list of positive integers \([a_1, a_2, \ldots, a_{n-1}]\) 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 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 \( \text{digraph} \).

Example

\[
\text{gap> D := Digraph([[2], [3], [2, 3]]);}
\text{<immutable digraph with 3 vertices, 4 edges>}
\text{gap> DigraphPath(D, 1, 3);}
\text{[ [ 1, 2, 3 ], [ 1, 1 ] ]}
\text{gap> DigraphPath(D, 2, 1);}
\text{fail}
\text{gap> DigraphPath(D, 3, 3);}
\text{[ [ 3, 3 ], [ 2 ] ]}
\text{gap> DigraphPath(D, 1, 1);}
\text{fail}
\]

5.3.19 \( \text{DigraphShortestPath} \)

\( \text{DigraphShortestPath} (\text{digraph}, u, v) \)

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 \( \text{DigraphPath} \) (5.3.18) 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 \( \text{digraph} \).

Example

\[
\text{gap> D := Digraph([[1, 2], [3], [2, 4], [1], [2, 4]]);}
\text{<immutable digraph with 5 vertices, 8 edges>}
\text{gap> DigraphShortestPath(D, 5, 1);}
\text{[ [ 5, 4, 1 ], [ 2, 1 ] ]}
\text{gap> DigraphShortestPath(D, 3, 3);}
\text{[ [ 3, 2, 3 ], [ 1, 1 ] ]}
\text{gap> DigraphShortestPath(D, 5, 5);}
\text{fail}
\text{gap> DigraphShortestPath(D, 1, 1);}
\text{[ [ 1, 1 ], [ 1 ] ]}
\]
5.3.20 IteratorOfPaths

\[\text{IteratorOfPaths(digraph, u, v)}\] (operation)

Returns: An iterator.

If `digraph` is a digraph or a list of adjacencies which defines a digraph - see 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 `digraph` from the vertex \(u\) to the vertex \(v\).

See DigraphPath (5.3.18) 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

\[
\begin{align*}
gap & D := \text{Digraph}([[1, 4, 4, 2], [3, 5], [2, 3], [1, 2], [4]]); \\
gap & \text{iter} := \text{IteratorOfPaths}(D, 1, 4); \\
gap & \text{NextIterator} \text{iter}; \\
& [ [ 1, 4 ], [ 2 ] ] \\
gap & \text{NextIterator} \text{iter}; \\
& [ [ 1, 4 ], [ 3 ] ] \\
gap & \text{NextIterator} \text{iter}; \\
& [ [ 1, 2, 5, 4 ], [ 4, 2, 1 ] ] \\
gap & \text{IsDoneIterator} \text{iter}; \\
& \text{true} \\
gap & \text{iter} := \text{IteratorOfPaths}(D, 4, 3); \\
& \langle \text{iterator} \rangle \\
gap & \text{NextIterator} \text{iter}; \\
& [ [ 4, 1, 2, 3 ], [ 1, 4, 1 ] ]
\end{align*}
\]

5.3.21 DigraphAllSimpleCircuits

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

Returns: A list of lists of vertices.

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

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 \((v_1, e_1, v_2, e_2, \ldots, e_{n-1}, v_n, e_n, v_1)\) 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

\[
\begin{align*}
gap & D := \text{Digraph}([[], [3], [2, 4], [5, 4], [4]]); \\
& \langle \text{immutable digraph with 5 vertices, 6 edges} \rangle \\
\end{align*}
\]

\[
\begin{align*}
gap & \text{DigraphAllSimpleCircuits}(D); \\
\end{align*}
\]
Digraphs

5.3.22 DigraphLongestSimpleCircuit

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

Returns: A list of vertices, or fail.

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

This attribute computes \text{DigraphAllSimpleCircuits} to find all the simple circuits of \text{digraph}, and returns one of maximal length. A simple circuit is represented as a list of vertices, in the same way as described in \text{DigraphAllSimpleCircuits} (5.3.21).

If \text{digraph} has no simple circuits, then this attribute returns fail. If \text{digraph} has multiple simple circuits of maximal length, then this attribute returns one of them.

\[\text{Example}\]

\[
\begin{array}{l}
\text{gap> D := Digraph([[1], [3], [2, 4], [5, 4], [4]]);;}
\text{gap> DigraphLongestSimpleCircuit(D); [ 4, 5 ]}
\text{gap> D := ChainDigraph(10);} \\text{(fail)}
\text{gap> D := Digraph([[3], [1], [1]]);} \\text{(fail)}
\text{gap> DigraphLongestSimpleCircuit(D); [ 1, 3 ]}
\text{gap> D := CycleDigraph(IsMutableDigraph, 3);} \\text{(fail)}
\text{gap> DigraphLongestSimpleCircuit(D); [ 1, 2, 3 ]}
\end{array}
\]

5.3.23 DigraphLayers

\[\text{DigraphLayers}(\text{digraph, vertex})\]

Returns: A list.

\[\text{Example}\]

\[
\begin{array}{l}
\text{gap> D := Digraph([[1], [3], [2, 4], [5, 4], [4]]);} \\text{(fail)}
\text{gap> DigraphLongestSimpleCircuit(D); [ 4, 5 ]}
\text{gap> D := ChainDigraph(10);} \\text{(fail)}
\text{gap> D := Digraph([[3], [1], [1], [1]]);} \\text{(fail)}
\text{gap> DigraphLongestSimpleCircuit(D); [ 1, 3 ]}
\text{gap> D := GeneralisedPetersenGraph(IsMutableDigraph, 4, 1);} \\text{(fail)}
\text{gap> DigraphLongestSimpleCircuit(D); [ 1, 2, 3, 4, 8, 7, 6, 5 ]}
\end{array}
\]
This operation returns a list \( \text{list} \) such that \( \text{list}[i] \) is the list of vertices whose minimum distance from the vertex \( \text{vertex} \) in \( \text{digraph} \) is \( i - 1 \). Vertex \( \text{vertex} \) is assumed to be at distance 0 from itself.

\[
\text{gap> } D := \text{CompleteDigraph}(4);;
\text{gap> } \text{DigraphLayers}(D, 1);
[ [ 1 ], [ 2, 3, 4 ] ]
\]

### 5.3.24 DigraphDegeneracy

\( \text{DigraphDegeneracy}(\text{digraph}) \)

**Returns:** A non-negative integer, or \( \text{fail} \).

If \( \text{digraph} \) is a symmetric digraph without multiple edges - see \text{IsSymmetricDigraph} (6.1.12) and \text{IsMultiDigraph} (6.1.10) - then this attribute returns the degeneracy of \( \text{digraph} \).

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 \( \text{fail} \).

\[
\text{gap> } D := \text{DigraphSymmetricClosure}(\text{ChainDigraph}(5));
\text{gap> } \text{DigraphDegeneracy}(D);
1
\text{gap> } D := \text{CompleteDigraph}(5);
\text{gap> } \text{DigraphDegeneracy}(D);
4
\text{gap> } D := \text{Digraph}(\text{[[1], [2, 4, 5], [3, 4], [2, 3, 4], [2], []]});
<\text{immutable digraph with 6 vertices, 10 edges}>
\text{gap> } \text{DigraphDegeneracy}(D);
1
\text{gap> } D := \text{GeneralisedPetersenGraph}(\text{IsMutableDigraph}, 10, 3);
<\text{mutable digraph with 20 vertices, 60 edges}>
\text{gap> } \text{DigraphDegeneracy}(D);
3
\]

### 5.3.25 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} (5.3.24) - 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 \( \text{fail} \).

\[
\text{gap> } D := \text{DigraphSymmetricClosure}(\text{ChainDigraph}(5));
\text{gap> } \text{DigraphDegeneracyOrdering}(D);
[ 5, 4, 3, 2, 1 ]
\]
gap> D := CompleteDigraph(5);;
gap> DigraphDegeneracyOrdering(D);
[ 5, 4, 3, 2, 1 ]
gap> D := Digraph([[1], [2, 4, 5], [3, 4], [2, 3, 4], [2], [ ]]);
<immutable digraph with 6 vertices, 10 edges>
gap> DigraphDegeneracyOrdering(D);
[ 1, 6, 5, 2, 4, 3 ]
gap> D := GeneralisedPetersenGraph(IsMutableDigraph, 3, 1);
<mutable digraph with 6 vertices, 18 edges>
gap> DigraphDegeneracyOrdering(D);
[ 6, 5, 4, 1, 3, 2 ]

5.3.26 HamiltonianPath

▷ HamiltonianPath(digraph)  (attribute)

Returns: A list or fail.
Returns a Hamiltonian path if one exists, fail if not.

A Hamiltonian path of a digraph with n vertices is directed cycle of length n. If digraph is a
digraph that contains a Hamiltonian path, then this function returns one, described in the form used by
DigraphAllSimpleCircuits (5.3.21). Note if digraph has 0 or 1 vertices, then HamiltonianPath returns [] or [1], respectively.

The method used in this attribute has the same worst case complexity as DigraphMonomorphism (7.3.4).

Example

| gap> D := Digraph([[]]);
| <immutable empty digraph with 1 vertex>
| gap> HamiltonianPath(D);
| [ 1 ]
| gap> D := Digraph([[2], [1]]);
| <immutable digraph with 2 vertices, 2 edges>
| gap> HamiltonianPath(D);
| [ 1, 2 ]
| gap> D := Digraph([[3], [], [2]]);
| <immutable digraph with 3 vertices, 2 edges>
| gap> HamiltonianPath(D);
| fail
| gap> D := Digraph([[2], [3], [1]]);
| <immutable digraph with 3 vertices, 3 edges>
| gap> HamiltonianPath(D);
| [ 1, 2, 3 ]
| gap> D := GeneralisedPetersenGraph(IsMutableDigraph, 5, 2);
| <mutable digraph with 10 vertices, 30 edges>
| gap> HamiltonianPath(D);
| fail

5.3.27 NrSpanningTrees

▷ NrSpanningTrees(digraph)  (attribute)

Returns: An integer.
Returns the number of spanning trees of the symmetric digraph \texttt{digraph}. \texttt{NrSpanningTrees} will return an error if \texttt{digraph} is not a symmetric digraph.

See \texttt{IsSymmetricDigraph (6.1.12)} and \texttt{IsUndirectedSpanningTree (4.1.2)} for more information.

Example

\begin{verbatim}
gap> D := CompleteDigraph(5);<immutable complete digraph with 5 vertices>gap> NrSpanningTrees(D);125gap> D := DigraphSymmetricClosure(CycleDigraph(24));;gap> NrSpanningTrees(D);24gap> NrSpanningTrees(EmptyDigraph(0));0gap> D := GeneralisedPetersenGraph(IsMutableDigraph, 9, 2);<mutable digraph with 18 vertices, 54 edges>gap> NrSpanningTrees(D);1134225
\end{verbatim}

5.4 Cayley graphs of groups

5.4.1 GroupOfCayleyDigraph

\begin{verbatim}
▷ GroupOfCayleyDigraph(digraph) (attribute)
▷ SemigroupOfCayleyDigraph(digraph) (attribute)

Returns: A group or semigroup.

If \texttt{digraph} is an immutable Cayley graph of a group \texttt{G} and \texttt{digraph} belongs to the category \texttt{IsCayleyDigraph (3.1.4)}, then \texttt{GroupOfCayleyDigraph} returns \texttt{G}.

If \texttt{digraph} is a Cayley graph of a semigroup \texttt{S} and \texttt{digraph} belongs to the category \texttt{IsCayleyDigraph (3.1.4)}, then \texttt{SemigroupOfCayleyDigraph} returns \texttt{S}.

See also \texttt{GeneratorsOfCayleyDigraph (5.4.2)}.
\end{verbatim}

Example

\begin{verbatim}
gap> G := DihedralGroup(IsPermGroup, 8);
Group([ (1,2,3,4), (2,4)]
gap> digraph := CayleyDigraph(G);
<immutable digraph with 8 vertices, 16 edges>gap> GroupOfCayleyDigraph(digraph) = G;
true
\end{verbatim}

5.4.2 GeneratorsOfCayleyDigraph

\begin{verbatim}
▷ GeneratorsOfCayleyDigraph(digraph) (attribute)

Returns: A list of generators.

If \texttt{digraph} is an immutable Cayley graph of a group or semigroup with respect to a set of generators \texttt{gens} and \texttt{digraph} belongs to the category \texttt{IsCayleyDigraph (3.1.4)}, then \texttt{GeneratorsOfCayleyDigraph} return the list of generators \texttt{gens} over which \texttt{digraph} is defined.

See also \texttt{GroupOfCayleyDigraph (5.4.1)} or \texttt{SemigroupOfCayleyDigraph (5.4.1)}.
\end{verbatim}
5.5 Associated semigroups

5.5.1 AsSemigroup (for a filter and a digraph)

\[ \text{AsSemigroup}(\text{filt}, \text{digraph}) \]

(operation)

\[ \text{AsMonoid}(\text{filt}, \text{digraph}) \]

(operation)

Returns: A semilattice of partial perms.

The operation AsSemigroup requires that filt be equal to IsPartialPermSemigroup (Reference: IsPartialPermSemigroup). If digraph is a IsJoinSemilatticeDigraph (6.1.17) or IsLatticeDigraph (6.1.17) 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 (6.1.17) but not IsJoinSemilatticeDigraph (6.1.17) 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 (6.1.17), 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

\[
\text{gap> } \text{di} := \text{Digraph}([[1], [1, 2], [1, 3], [1, 4], [1, 2, 3, 5]]);
\text{<immutable digraph with 5 vertices, 11 edges>}
\text{gap> } S := \text{AsSemigroup}('IsPartialPermSemigroup', \text{di});
\text{<partial perm semigroup of rank 3 with 4 generators>}
\text{gap> } \text{ForAll(Elements(S), IsIdempotent)};
\text{true}
\text{gap> } \text{IsInverseSemigroup}(S);
\text{true}
\text{gap> } \text{Size}(S);
5
\text{gap> } \text{di} := \text{Digraph}([[1], [1, 2], [1, 2, 3]]);
\text{<immutable digraph with 3 vertices, 6 edges>}
\text{gap> } M := \text{AsMonoid}('IsPartialPermMonoid', \text{di});
\]
5.5.2  AsSemigroup (for a filter, semilattice digraph, and two lists)

\textbf{AsSemigroup}(filt, Y, gps, homs)

\textit{Returns:} A Clifford semigroup of partial perms.

The operation \textit{AsSemigroup} requires that \textit{filt} be equal to \textit{IsPartialPermSemigroup} (Reference: \textit{IsPartialPermSemigroup}). If \textit{Y} is a \textit{IsJoinSemilatticeDigraph} (6.1.17) or \textit{IsMeetSemilatticeDigraph} (6.1.17), \textit{gps} is a list of groups corresponding to each vertex, and \textit{homs} is a list containing for each edge \((i, j)\) in the transitive reduction of \textit{digraph} a triple \([i, j, \text{hom}]\) where \text{hom} is a group homomorphism from \textit{gps}[i] to \textit{gps}[j], and the diagram of homomorphisms commutes, then \textit{AsSemigroup} returns a semigroup of partial perms which is isomorphic to the strong semilattice of groups \(S[Y;gps;homs]\).

\begin{verbatim}
<partial perm monoid of rank 2 with 3 generators>
gap> Size(M); 3

gap> AsSemigroup(IsPartialPermSemigroup, gr, [G1, G2, G3], 
[ [1, 3, hom13], [2, 3, hom23] ]); 20

gap> D := GreensDClasses(T);;
gap> List(D, x -> Size(x));[ 6, 12, 2 ]
\end{verbatim}

5.6  Planarity

5.6.1  KuratowskiPlanarSubdigraph

\textbf{KuratowskiPlanarSubdigraph}(digraph)

\textit{Returns:} A list or fail.

\textbf{KuratowskiPlanarSubdigraph} returns the immutable list of lists of out-neighbours of a (not necessarily induced) subdigraph of the digraph \textit{digraph} that witnesses the fact that \textit{digraph} is not planar, or fail if \textit{digraph} is planar. In other words, \textbf{KuratowskiPlanarSubdigraph} returns the out-neighbours of a subdigraph of \textit{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 digraph are ignored when considering whether or not digraph is planar.

See also IsPlanarDigraph (6.4.1) and SubdigraphHomeomorphicToK33 (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
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]]);
<immutable digraph with 11 vertices, 25 edges>
gap> KuratowskiPlanarSubdigraph(D);
fail

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]]);
<immutable digraph with 10 vertices, 50 edges>
gap> IsPlanarDigraph(D);
false

5.6.2 KuratowskiOuterPlanarSubdigraph

KuratowskiOuterPlanarSubdigraph(digraph)

Returns: A list or fail.

KuratowskiOuterPlanarSubdigraph returns the immutable list of immutable lists of out-neighbours of a (not necessarily induced) subdigraph of the digraph digraph that witnesses the fact that digraph is not outer planar, or fail if digraph is outer planar. In other words, KuratowskiOuterPlanarSubdigraph returns the out-neighbours of a subdigraph of 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 digraph are ignored when considering whether or not 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]]);
<immutable digraph with 11 vertices, 25 edges>
gap> KuratowskiOuterPlanarSubdigraph(D);
[ [ 3, 5, 10 ], [ 9, 8, 10 ], [ 4 ], [ 6 ], [ 11 ], [ 7 ], [ 8 ],
  [ ], [ 11 ], [ ], [ ] ]
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], [2, 3, 5, 6, 7, 8], [3, 5]]);
<immutable digraph with 10 vertices, 50 edges>
gap> IsOuterPlanarDigraph(D);
false
```

5.6.3 PlanarEmbedding

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

(\text{attribute})

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

If \text{digraph} is a planar digraph, then PlanarEmbedding returns the immutable list of lists 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 planar, then \text{fail} is returned.

The directions and multiplicities of any edges in \text{digraph} are ignored by PlanarEmbedding. See also \text{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]]);
<immutable digraph with 11 vertices, 25 edges>
gap> PlanarEmbedding(D);
```
Digraphs

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]]);
<immutable digraph with 10 vertices, 50 edges>
gap> PlanarEmbedding(D);
fail

gap> D := Digraph(IsMutableDigraph, [[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]]);
<mutable digraph with 11 vertices, 25 edges>
gap> PlanarEmbedding(D);
fail

gap> OuterPlanarEmbedding(D);
fail

gap> D := Digraph(IsMutableDigraph, [[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]]);
<mutable digraph with 11 vertices, 25 edges>
gap> PlanarEmbedding(D);
fail

5.6.4 OuterPlanarEmbedding

OuterPlanarEmbedding(digraph) (attribute)

Returns: A list or fail.

If digraph is an outer planar digraph, then OuterPlanarEmbedding returns the immutable list of lists of out-neighbours of a subdigraph of digraph such that each vertex's neighbours are given in clockwise order. If digraph is not outer planar, then fail is returned.

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

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> 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]]);
<immutable digraph with 11 vertices, 25 edges>
gap> OuterPlanarEmbedding(D);
fail

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]]);
<immutable digraph with 10 vertices, 50 edges>
gap> OuterPlanarEmbedding(D);
fail

gap> OuterPlanarEmbedding(CompleteBipartiteDigraph(2, 2));
[ [ 3, 4 ], [ 4, 3 ], [ ] ]
gap> D := Digraph(IsMutableDigraph, [[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]]);
<mutable digraph with 11 vertices, 25 edges>
5.6.5 SubdigraphHomeomorphicToK23

These attributes return the immutable list of lists of out-neighbours of a subdigraph of the digraph `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 `digraph` has no such subdigraphs, then `fail` is returned.

See also `IsPlanarDigraph (6.4.1)` and `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].
Digraphs

```
gap> SubdigraphHomeomorphicToK4(CompleteDigraph(3));
fail
gap> D := Digraph(IsMutableDigraph, [[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]]);
<mutable digraph with 11 vertices, 25 edges>
gap> SubdigraphHomeomorphicToK4(D);
[ [ 3, 5, 10 ], [ 9, 8, 10 ], [ 4 ], [ 6 ], [ 7, 11 ], [ 7 ], [ 8 ],
  [ ], [ 11 ], [ ], [ ] ]
gap> SubdigraphHomeomorphicToK23(D);
[ [ 3, 5, 10 ], [ 9, 8, 10 ], [ 4 ], [ 6 ], [ 11 ], [ 7 ], [ 8 ],
  [ ], [ 11 ], [ ], [ ] ]
gap> D := Digraph(IsMutableDigraph, [[3, 5, 10], [8, 9, 10], [1, 4],
> [3, 6], [1, 11], [4, 7], [6, 8], [2, 7], [2, 11], [1, 2], [5, 9]]);
<mutable digraph with 11 vertices, 24 edges>
gap> SubdigraphHomeomorphicToK4(D);
fail
gap> SubdigraphHomeomorphicToK23(D);
fail
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

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

Returns: true or false.

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.

If the argument \text{digraph} is mutable, then the return value of this property is recomputed every time it is called.

Example

\begin{verbatim}
gap> D := Digraph([[1, 2], [2]]);
<immutable digraph with 2 vertices, 3 edges>
gap> DigraphEdges(D);
[ [ 1, 1 ], [ 1, 2 ], [ 2, 2 ] ]
gap> DigraphHasLoops(D);
true
gap> D := Digraph([[2, 3], [1], [2]]);
<immutable digraph with 3 vertices, 4 edges>
gap> DigraphEdges(D);
[ [ 1, 2 ], [ 1, 3 ], [ 2, 1 ], [ 3, 2 ] ]
gap> DigraphHasLoops(D);
false
gap> D := CompleteDigraph(IsMutableDigraph, 4);
<mutable digraph with 4 vertices, 12 edges>
gap> DigraphHasLoops(D);
false
\end{verbatim}

6.1.2 IsAntiSymmetricDigraph

\begin{verbatim}
\text{IsAntiSymmetricDigraph}(\text{digraph})
\end{verbatim}

Returns: true or false.

This property is 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.
If the argument `digraph` is mutable, then the return value of this property is recomputed every time it is called.

```
gap> digraph := Digraph([[2], [1, 3], [2, 3]]);
gap> IsAntisymmetricDigraph(digraph);  
false
gap> DigraphEdges(digraph){[1, 2]};
[ [ 1, 2 ], [ 2, 1 ] ]
gap> digraph := Digraph([[1, 2], [3, 3], [1]]);
gap> IsAntisymmetricDigraph(digraph);
true

6.1.3 IsBipartiteDigraph

```

```

6.1.4 IsCompleteBipartiteDigraph

```
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).

If the argument $digraph$ is mutable, then the return value of this property is recomputed every time it is called.

Example

```gap
gap> D := CycleDigraph(2);
<immutable cycle digraph with 2 vertices>
gap> IsCompleteBipartiteDigraph(D);
true
gap> D := CycleDigraph(4);
<immutable cycle digraph with 4 vertices>
gap> IsBipartiteDigraph(D);
true
gap> IsCompleteBipartiteDigraph(D);
false
gap> D := CompleteBipartiteDigraph(IsMutableDigraph, 5, 4);
<mutable digraph with 9 vertices, 40 edges>
gap> IsCompleteBipartiteDigraph(D);
true
```

### 6.1.5 IsCompleteDigraph

**Example**

```gap
gap> D := Digraph([[2, 3], [1, 3], [1, 2]]);
<immutable digraph with 3 vertices, 6 edges>
gap> IsCompleteDigraph(D);
true
gap> D := Digraph([[2, 2], [1]]);
<immutable multidigraph with 2 vertices, 3 edges>
gap> IsCompleteDigraph(D);
false
gap> D := CompleteBipartiteDigraph(IsMutableDigraph, 5, 4);
<mutable digraph with 9 vertices, 40 edges>
gap> IsCompleteDigraph(D);
false
```

### 6.1.6 IsCompleteMultipartiteDigraph

**Example**

```gap
gap> D := CompleteBipartiteDigraph(IsMutableDigraph, 5, 4);
<mutable digraph with 9 vertices, 40 edges>
gap> IsCompleteBipartiteDigraph(D);
true
```

This property returns true if $digraph$ is a complete multipartite digraph, and false if not.
A digraph is a *complete multipartite digraph* if and only if its vertices can be partitioned into at least two maximal independent sets, where every possible edge between these independent sets occurs in the digraph exactly once.

If the argument `digraph` is mutable, then the return value of this property is recomputed every time it is called.

```gap
gap> D := CompleteMultipartiteDigraph([2, 4, 6]);
<immutable complete multipartite digraph with 12 vertices, 88 edges>
gap> IsCompleteMultipartiteDigraph(D);
true
gap> D := CompleteBipartiteDigraph(IsMutableDigraph, 5, 4);
<mutable digraph with 9 vertices, 40 edges>
gap> IsCompleteMultipartiteDigraph(D);
true
```

**6.1.7 IsEmptyDigraph**

*IsEmptyDigraph(digraph)* (property)

Returns: true or false.

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

`IsNullDigraph` is a synonym for `IsEmptyDigraph`.

If the argument `digraph` is mutable, then the return value of this property is recomputed every time it is called.

```gap
gap> D := Digraph([[], []]);
<immutable empty digraph with 2 vertices>
gap> IsEmptyDigraph(D);
true
gap> IsNullDigraph(D);
true
gap> D := Digraph([[], [1]]);
<immutable digraph with 2 vertices, 1 edge>
gap> IsEmptyDigraph(D);
false
gap> IsNullDigraph(D);
false
```

**6.1.8 IsEquivalenceDigraph**

*IsEquivalenceDigraph(digraph)* (property)

Returns: true or false.

A digraph is an equivalence digraph if and only if the digraph satisfies all of `IsReflexiveDigraph` *(6.1.11)*, `IsSymmetricDigraph` *(6.1.12)* and `IsTransitiveDigraph` *(6.1.14)*. A partial order `digraph` corresponds to an equivalence relation.

If the argument `digraph` is mutable, then the return value of this property is recomputed every time it is called.
Digraphs

Example

gap> D := Digraph([[1, 3], [2], [1, 3]]);
<immutable digraph with 3 vertices, 5 edges>
gap> IsEquivalenceDigraph(D);
true

6.1.9 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.

If the argument digraph is mutable, then the return value of this property is recomputed every time it is called.

Example

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

6.1.10 IsMultiDigraph

▷ IsMultiDigraph(digraph)

Returns: true or false.

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

If the argument digraph is mutable, then the return value of this property is recomputed every time it is called.

Example

gap> D := Digraph("
a", "b", "c"], ["a", "b", "b"], ["b", "c", "a");
<immutable digraph with 3 vertices, 3 edges>
gap> IsMultiDigraph(D);
false
gap> D := DigraphFromDigraph6String("&Bug");
<immutable digraph with 3 vertices, 6 edges>
gap> IsDuplicateFree(DigraphEdges(D));
true
gap> IsMultiDigraph(D);
false
gap> D := Digraph([[1, 2, 3, 2], [2, 1], [3]]);
<immutable multidigraph with 3 vertices, 7 edges>
gap> IsDuplicateFree(DigraphEdges(D));
false
Digraphs

gap> IsMultiDigraph(D);
true
gap> D := DigraphMutableCopy(D);
<mutable multidigraph with 3 vertices, 7 edges>
gap> IsMultiDigraph(D);
true

6.1.11 IsReflexiveDigraph

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

\text{(property)}

\textbf{Returns:} true or false.

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

If the argument \textit{digraph} is mutable, then the return value of this property is recomputed every time it is called.

\begin{verbatim}
gap> D := Digraph([[1, 2], [2]]);
<immutable digraph with 2 vertices, 3 edges>
gap> IsReflexiveDigraph(D);
true
gap> D := Digraph([[3, 1], [4, 2], [3], [2, 1]]);
<immutable digraph with 4 vertices, 7 edges>
gap> IsReflexiveDigraph(D);
false
\end{verbatim}

6.1.12 IsSymmetricDigraph

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

\text{(property)}

\textbf{Returns:} true or false.

This property is true if the digraph \textit{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 \text{\texttt{AdjacencyMatrix(5.2.1)}}.

If the argument \textit{digraph} is mutable, then the return value of this property is recomputed every time it is called.

\begin{verbatim}
Example

gap> gr1 := Digraph([[2], [1, 3], [2, 3]]);
<immutable digraph with 3 vertices, 5 edges>
gap> IsSymmetricDigraph(gr1);
true
gap> adj1 := AdjacencyMatrix(gr1);
gap> Display(adj1);
[ [ 0, 1, 0 ],
[ 1, 0, 1 ],
[ 0, 1, 1 ] ]
gap> adj1 = TransposedMat(adj1);
true
gap> gr1 = DigraphReverse(gr1);
true
\end{verbatim}
Digraphs

6.1.13 IsTournament

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

\textbf{Returns:} true or false.

This property is true if the digraph \textit{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.

If the argument \textit{digraph} is mutable, then the return value of this property is recomputed every time it is called.

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

gap> D := Digraph([[2], [1], [3]]);
<immutable digraph with 3 vertices, 3 edges>
gap> IsTournament(D);
false

gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
gap> IsTournament(D);
true

gap> DigraphRemoveEdge(D, 1, 2);
<mutable digraph with 3 vertices, 2 edges>
gap> IsTournament(D);
false
\end{verbatim}

6.1.14 IsTransitiveDigraph

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

\textbf{Returns:} true or false.

This property is true if the digraph \textit{digraph} is transitive, and false if it is not. A digraph is transitive if whenever \([i, j]\) and \([j, k]\) are edges of the digraph, then \([i, k]\) is also an edge of the digraph.

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

If the argument \textit{digraph} is mutable, then the return value of this property is recomputed every time it is called.

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

\subsection{IsPreorderDigraph}

\begin{verbatim}
▷ IsPreorderDigraph(digraph) \hspace{5cm} (property)
▷ IsQuasiorderDigraph(digraph) \hspace{5cm} (property)

Returns: true or false.

A digraph is a preorder digraph if and only if the digraph satisfies both IsReflexiveDigraph (6.1.11) and IsTransitiveDigraph (6.1.14). A preorder digraph (or quasiorder digraph) \textit{digraph} corresponds to the preorder relation $\leq$ defined by $x \leq y$ if and only if $[x, y]$ is an edge of \textit{digraph}.

If the argument \textit{digraph} is mutable, then the return value of this property is recomputed every time it is called.

\begin{verbatim}
gap> D := Digraph([[1, 2, 3], [3], [3]]);  <immutable digraph with 3 vertices, 5 edges> gap> IsPreorderDigraph(D);  true gap> D := Digraph([[1 .. 4], [1 .. 4], [1 .. 4], [1 .. 4]]);  <immutable digraph with 4 vertices, 16 edges> gap> IsPreorderDigraph(D);  true gap> D := Digraph([[2], [3], [4], [5], [1]]);  <immutable digraph with 5 vertices, 5 edges> gap> IsPreorderDigraph(D);  true gap> D := Digraph([[1], [1, 2], [2, 3]]);  <immutable digraph with 3 vertices, 5 edges> gap> IsQuasiorderDigraph(D);  false
\end{verbatim}
6.1.16  IsPartialOrderDigraph

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

\textbf{Returns:} true or false.

A digraph is a partial order digraph if and only if the digraph satisfies all of IsReflexiveDigraph (6.1.11), IsAntisymmetricDigraph (6.1.2) and IsTransitiveDigraph (6.1.14). 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.

If the argument \text{digraph} is mutable, then the return value of this property is recomputed every time it is called.

\begin{verbatim}
gap> D := Digraph([[1, 3], [2, 3], [3]]);
gep> IsPartialOrderDigraph(D);
true
gap> D := CycleDigraph(5);
gep> IsPartialOrderDigraph(D);
false
gap> D := Digraph([[1, 1], [1, 1, 2], [3], [3, 3, 4, 4]]);
gep> IsPartialOrderDigraph(D);
true
\end{verbatim}

6.1.17  IsMeetSemilatticeDigraph

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

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

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

\textbf{Returns:} true or false.

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

For a partial order digraph \text{IsPartialOrderDigraph} (6.1.16) 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 \textit{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 \textit{join semilattice} is a partial order where every pair of vertices has a least upper bound (join) with respect to the relation.

If the argument \text{digraph} is mutable, then the return value of this property is recomputed every time it is called.

\begin{verbatim}
gap> D := Digraph([[1, 3], [2, 3], [3]]);
gep> IsMeetSemilatticeDigraph(D);
false
gap> IsJoinSemilatticeDigraph(D);
true
gap> IsLatticeDigraph(D);
false
gap> D := Digraph([[1], [2], [1 .. 3]]);
\end{verbatim}
6.2 Regularity

6.2.1 IsInRegularDigraph

▷ IsInRegularDigraph(digraph)  
\text{(property)}

\textbf{Returns:} true or false.

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

If the argument \textit{digraph} is mutable, then the return value of this property is recomputed every time it is called.

Example

\begin{verbatim}
gap> IsInRegularDigraph(CompleteDigraph(4));
true
\end{verbatim}

6.2.2 IsOutRegularDigraph

▷ IsOutRegularDigraph(digraph)  
\text{(property)}

\textbf{Returns:} true or false.

This property is true if there is an integer n such that for every vertex v of digraph \textit{digraph} there are exactly n edges starting at v.

See also IsInRegularDigraph (6.2.1) and IsRegularDigraph (6.2.3).
If the argument `digraph` is mutable, then the return value of this property is recomputed every time it is called.

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| 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)`.

If the argument `digraph` is mutable, then the return value of this property is recomputed every time it is called.

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| 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.

If the argument `digraph` is mutable, then the return value of this property is recomputed every time it is called.

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| gap> D := DigraphSymmetricClosure(ChainDigraph(5));
  false |
| gap> IsDistanceRegularDigraph(D);
  true |
| gap> D := Digraph([2, 3, 4], [1, 3, 4], [1, 2, 4], [1, 2, 3]);
  <immutable digraph with 4 vertices, 12 edges>
| gap> IsDistanceRegularDigraph(D);
  true |
6.3 Connectivity and cycles

6.3.1 IsAcyclicDigraph

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

**Returns:** true or false.

This property is true if the digraph \texttt{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.

If the argument \texttt{digraph} is mutable, then the return value of this property is recomputed every time it is called.

```
gap> Petersen := Graph(SymmetricGroup(5), [[1, 2]], OnSets, > function(x, y) > return IsEmpty(Intersection(x, y)); > end);;
gap> D := Digraph(Petersen);
<immutable digraph with 10 vertices, 30 edges>
gap> IsAcyclicDigraph(D);
false
```

```
gap> D := DigraphFromDiSparse6String(
> ".b_OGC1DBaPGkULEbQHCeRIdrHcuZMfFrDAbPhTi|zF");
<immutable digraph with 35 vertices, 34 edges>
gap> IsAcyclicDigraph(D);
true
```

```
gap> D := CompleteDigraph(IsMutableDigraph, 4);
<mutable digraph with 4 vertices, 12 edges>
gap> IsAcyclicDigraph(D);
false
```

```
gap> IsAcyclicDigraph(CycleDigraph(10));
true
```

```
gap> D := CompleteDigraph(IsMutableDigraph, 4);
<mutable digraph with 4 vertices, 12 edges>
gap> IsAcyclicDigraph(D);
false
```

6.3.2 IsChainDigraph

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

**Returns:** true or false.

IsChainDigraph returns true if the digraph \texttt{digraph} is isomorphic to the chain digraph with the same number of vertices as \texttt{digraph}, and false if it is not; see \texttt{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 \texttt{IsDirectedTree}(6.3.7) and \texttt{OutDegrees}(5.2.8).

If the argument \texttt{digraph} is mutable, then the return value of this property is recomputed every time it is called.

```
gap> D := Digraph([[1, 3], [2, 3], [3]]);
<immutable digraph with 3 vertices, 5 edges>
gap> IsChainDigraph(D);
false
```
6.3.3 IsConnectedDigraph

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

Returns: true or false.

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

If the argument \textit{digraph} is mutable, then the return value of this property is recomputed every time it is called.

\[
\begin{align*}
gap> & \text{D := Digraph([[2], [3], []]);;} \\
gap> & \text{IsConnectedDigraph(D);;} \\
gap> & \text{true} \\
gap> & \text{D := Digraph([[1, 3], [4], [3], []]);;} \\
gap> & \text{IsConnectedDigraph(D);;} \\
gap> & \text{false} \\
gap> & \text{D := Digraph(IsMutableDigraph, [[2], [3], []]);;} \\
gap> & \text{IsConnectedDigraph(D);;} \\
gap> & \text{true} \\
gap> & \text{D := Digraph(IsMutableDigraph, [[1, 3], [4], [3], []]);;} \\
gap> & \text{IsConnectedDigraph(D);;} \\
gap> & \text{false}
\end{align*}
\]

6.3.4 IsBiconnectedDigraph

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

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 \textit{digraph} is
Digraphs

biconnected, and false if it is not. In particular, IsBiconnectedDigraph returns false 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 ArticulationPoints (5.3.14).

If the argument digraph is mutable, then the return value of this property is recomputed every time it is called.

Example

```gap
gap> IsBiconnectedDigraph(Digraph([[1, 3], [2, 3], [3]]));
false
gap> IsBiconnectedDigraph(CycleDigraph(5));
true
gap> D := Digraph([[1, 1], [1, 1, 2], [3], [3, 3, 4, 4]]);
gap> IsBiconnectedDigraph(D);  # false
gap> D := CompleteBipartiteDigraph(IsMutableDigraph, 5, 4);
<mutable digraph with 9 vertices, 40 edges>
gap> IsBiconnectedDigraph(D);  # true
```

6.3.5 IsStronglyConnectedDigraph

▶ IsStronglyConnectedDigraph(digraph) (property)

**Returns:** true or false.

This property is true if the digraph digraph is strongly connected and false if it is not.

A digraph digraph is **strongly connected** if there is a directed path from every vertex to every other vertex. 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.

If the argument digraph is mutable, then the return value of this property is recomputed every time it is called.

Example

```gap
gap> D := CycleDigraph(250000);
<immutable cycle digraph with 250000 vertices>
gap> IsStronglyConnectedDigraph(D);  # true
gap> D := DigraphRemoveEdges(D, [[250000, 1]]);
<immutable digraph with 250000 vertices, 249999 edges>
gap> IsStronglyConnectedDigraph(D);  # false
gap> D := CycleDigraph(IsMutableDigraph, 250000);
<mutable digraph with 250000 vertices, 250000 edges>
gap> IsStronglyConnectedDigraph(D);  # true
gap> DigraphRemoveEdge(D, [250000, 1]);
<mutable digraph with 250000 vertices, 249999 edges>
gap> IsStronglyConnectedDigraph(D);  # false
```
6.3.6 IsAperiodicDigraph

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

**Returns:** true or false.

This property is true if the digraph \text{digraph} is aperiodic, i.e. if its \text{DigraphPeriod (5.3.15)} is equal to 1. Otherwise, the property is false.

If the argument \text{digraph} is mutable, then the return value of this property is recomputed every time it is called.

\begin{verbatim}
gap> D := Digraph([[6], [1], [2], [3], [4, 4], [5]]);
<immutable multidigraph with 6 vertices, 7 edges>
gap> IsAperiodicDigraph(D);
false

gap> D := Digraph([[2], [3, 5], [4], [5], [1, 2]]);
<immutable digraph with 5 vertices, 7 edges>
gap> IsAperiodicDigraph(D);
true

gap> D := Digraph(IsMutableDigraph, [[2], [3, 5], [4], [5], [1, 2]]);
<mutable digraph with 5 vertices, 7 edges>
gap> IsAperiodicDigraph(D);
true
\end{verbatim}

6.3.7 IsDirectedTree

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

**Returns:** true or false.

Returns true if the digraph \text{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 \text{DigraphSources (5.1.6)}.

If the argument \text{digraph} is mutable, then the return value of this property is recomputed every time it is called.

\begin{verbatim}
gap> D := Digraph([], [2]);
<immutable digraph with 2 vertices, 1 edge>
gap> IsDirectedTree(D);
false

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

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

gap> D := Digraph([[2, 3], [6], [4, 5], [], [], []]);
<immutable digraph with 6 vertices, 5 edges>
gap> IsDirectedTree(D);
true
\end{verbatim}
6.3.8 IsUndirectedTree

- `IsUndirectedTree(digraph)` (property)
- `IsUndirectedForest(digraph)` (property)

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` (6.1.12) 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` (5.3.9) and `DigraphDisjointUnion` (3.3.27). 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.

If the argument `digraph` is mutable, then the return value of this property is recomputed every time it is called.

Example

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

6.3.9 IsEulerianDigraph

- `IsEulerianDigraph(digraph)` (property)

Returns: true or false.

This property returns true if the digraph `digraph` is Eulerian.

A connected 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. Note that the empty digraph with at most one vertex is considered to be Eulerian.
If the argument \texttt{digraph} is mutable, then the return value of this property is recomputed every time it is called.

\begin{verbatim}
gap> D := Digraph([[1]]);
<immutable empty digraph with 1 vertex>
gap> IsEulerianDigraph(D);
true
gap> D := Digraph([[2], []]);
<immutable digraph with 2 vertices, 1 edge>
gap> IsEulerianDigraph(D);
false
gap> D := Digraph([[3], [], [2]]);
<immutable digraph with 3 vertices, 2 edges>
gap> IsEulerianDigraph(D);
false
gap> D := Digraph([[2], [3], [1]]);
<immutable digraph with 3 vertices, 3 edges>
gap> IsEulerianDigraph(D);
true
\end{verbatim}

\subsection*{6.3.10 \texttt{IsHamiltonianDigraph}}

\texttt{IsHamiltonianDigraph} (\texttt{digraph})

\textbf{Returns:} true or false.

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

A digraph with \( n \) vertices is \textit{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 \texttt{DigraphMonomorphism} (7.3.4).

If the argument \texttt{digraph} is mutable, then the return value of this property is recomputed every time it is called.

\begin{verbatim}
gap> g := Digraph([[1]]);
<immutable empty digraph with 1 vertex>
gap> IsHamiltonianDigraph(g);
true
gap> g := Digraph([[2], [1]]);
<immutable digraph with 2 vertices, 2 edges>
gap> IsHamiltonianDigraph(g);
true
gap> g := Digraph([[3], [], [2]]);
<immutable digraph with 3 vertices, 2 edges>
gap> IsHamiltonianDigraph(g);
false
gap> g := Digraph([[2], [3], [1]]);
<immutable digraph with 3 vertices, 3 edges>
gap> IsHamiltonianDigraph(g);
true
\end{verbatim}
6.3.11  IsCycleDigraph

\texttt{IsCycleDigraph(digraph)}  
\texttt{(property)}

\textbf{Returns:} true or false.

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

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

If the argument \textit{digraph} is mutable, then the return value of this property is recomputed every time it is called.

\begin{verbatim}
gap> D := Digraph([[1, 3], [2, 3], [3]]);  
<immutable digraph with 3 vertices, 5 edges>  
gap> IsCycleDigraph(D);  
false  
gap> D := CycleDigraph(5);  
<immutable cycle digraph with 5 vertices>  
gap> IsCycleDigraph(D);  
true  
gap> D := OnDigraphs(D, (1, 2, 3));  
<immutable digraph with 5 vertices, 5 edges>  
gap> D = CycleDigraph(5);  
false  
gap> IsCycleDigraph(D);  
true
\end{verbatim}

6.4  Planarity

6.4.1  IsPlanarDigraph

\texttt{IsPlanarDigraph(digraph)}  
\texttt{(property)}

\textbf{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 \textit{digraph} is planar and false if it is not. The directions and multiplicities of any edges in \textit{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].

\begin{verbatim}
gap> IsPlanarDigraph(CompleteDigraph(4));  
true  
gap> IsPlanarDigraph(CompleteDigraph(5));  
false  
gap> IsPlanarDigraph(CompleteBipartiteDigraph(2, 3));  
true  
gap> IsPlanarDigraph(CompleteBipartiteDigraph(3, 3));  
false
\end{verbatim}
Digraphs

6.4.2 IsOuterPlanarDigraph

\texttt{IsOuterPlanarDigraph(digraph)}

\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}
> IsOuterPlanarDigraph(CompleteDigraph(4));
false
> IsOuterPlanarDigraph(CompleteDigraph(5));
false
> IsOuterPlanarDigraph(CompleteBipartiteDigraph(2, 3));
false
> IsOuterPlanarDigraph(CompleteBipartiteDigraph(3, 3));
false
> IsOuterPlanarDigraph(CycleDigraph(10));
true
> IsOuterPlanarDigraph(CompleteDigraph(IsMutableDigraph, 4));
false
> IsOuterPlanarDigraph(CompleteDigraph(IsMutableDigraph, 5));
false
> IsOuterPlanarDigraph(CompleteBipartiteDigraph(IsMutableDigraph, 2, 3));
false
> IsOuterPlanarDigraph(CompleteBipartiteDigraph(IsMutableDigraph, 3, 3));
false
> IsOuterPlanarDigraph(CycleDigraph(IsMutableDigraph, 10));
true
\end{verbatim}
6.5 Homomorphisms and transformations

6.5.1 IsDigraphCore

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

\textbf{Returns:} true or false.

This property returns true if \text{digraph} is a core, and false if it is not.

A digraph \( D \) is a core if and only if it has no proper subdigraphs \( A \) such that there exists a homomorphism from \( D \) to \( A \). In other words, a digraph \( D \) is a core if and only if every endomorphism on \( D \) is an automorphism on \( D \).

If the argument \text{digraph} is mutable, then the return value of this property is recomputed every time it is called.

Example

\begin{verbatim}
gap> D := CompleteDigraph(6);<immutable complete digraph with 6 vertices>
gap> IsDigraphCore(D);true
gap> D := DigraphSymmetricClosure(CycleDigraph(6));<immutable symmetric digraph with 6 vertices, 12 edges>
gap> DigraphHomomorphism(D, CompleteDigraph(2));Transformation([1, 2, 1, 2, 1, 2])
gap> IsDigraphCore(D);false
\end{verbatim}

6.5.2 IsEdgeTransitive

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

\textbf{Returns:} true or false.

If \text{digraph} is a digraph without multiple edges, then \text{IsEdgeTransitive} returns true if \text{digraph} is edge transitive, and false otherwise. A digraph is edge transitive if its automorphism group acts transitively on its edges (via the action \text{OnPairs} (Reference: \text{OnPairs})).

If the argument \text{digraph} is mutable, then the return value of this property is recomputed every time it is called.

Example

\begin{verbatim}
gap> IsEdgeTransitive(CompleteDigraph(2));true
gap> IsEdgeTransitive(ChainDigraph(3));false
gap> IsEdgeTransitive(Digraph([[2], [3, 3, 3], []]));Error, the argument <D> must be a digraph with no multiple edges,
\end{verbatim}

6.5.3 IsVertexTransitive

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

\textbf{Returns:} true or false.

If \text{digraph} is a digraph, then \text{IsVertexTransitive} returns true if \text{digraph} is vertex transitive, and false otherwise. A digraph is vertex transitive if its automorphism group acts transitively on its vertices.

If the argument \text{digraph} is mutable, then the return value of this property is recomputed every time it is called.
Example

```
gap> IsVertexTransitive(CompleteDigraph(2));
true
gap> IsVertexTransitive(ChainDigraph(3));
false
```
Chapter 7

Homomorphisms

7.1 Acting on digraphs

7.1.1 OnDigraphs (for a digraph and a perm)

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

Returns: A digraph.

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

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

If \text{digraph} belongs to \text{IsMutableDigraph} (3.1.2), then relabelling of the vertices is performed directly on \text{digraph}. If \text{digraph} belongs to \text{IsImmutableDigraph} (3.1.3), an immutable copy of \text{digraph} with the vertices relabelled is returned.

Example

\[
\begin{array}{l}
gap> D := \text{Digraph}([[3], [1, 3, 5], [1], [1, 2, 4], [2, 3, 5]]); \\
<\text{immutable digraph with 5 vertices, 11 edges}> \\
gap> \text{new} := \text{OnDigraphs}(D, (1, 2)); \\
<\text{immutable digraph with 5 vertices, 11 edges}> \\
gap> \text{OutNeighbours(new)}; \\
[ [ 2, 3, 5 ], [ 3 ], [ 2 ], [ 2, 1, 4 ], [ 1, 3, 5 ] ] \\
gap> D := \text{Digraph}([[2], [], [2]]); \\
<\text{immutable digraph with 3 vertices, 2 edges}> \\
gap> t := \text{Transformation}([[1, 2, 1]]); \\
gap> \text{new} := \text{OnDigraphs}(D, t); \\
<\text{immutable multidigraph with 3 vertices, 2 edges}> \\
gap> \text{OutNeighbours(new)}; \\
[ [ 2, 2 ], [ ], [ ] ]
\end{array}
\]
7.1.2 OnMultiDigraphs

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

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

**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`. If `digraph` belongs to `IsMutableDigraph(3.1.2)`, then relabelling of the vertices is performed directly on `digraph`. If `digraph` belongs to `IsImmutableDigraph(3.1.3)`, an immutable copy of `digraph` with the vertices relabelled is returned.

**Example**

```gap
gap> D1 := Digraph([ [3, 6, 3], [], [3], [9, 10], [9], [], [10, 4, 10], [], []];
<immutable multidigraph with 10 vertices, 10 edges>

gap> p := BlissCanonicalLabelling(D1);
[ (1,7)(3,6)(4,10)(5,9), () ]

gap> D2 := OnMultiDigraphs(D1, p);
<immutable multidigraph with 10 vertices, 10 edges>

gap> OutNeighbours(D2);
[ [ ], [ ], [ ], [ ], [ ], [6], [6,3,6], [4,10,4], [5], [5,4] ]
```

7.2 Isomorphisms and canonical labelings

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

7.2.1 DigraphsUseNauty

\[\text{DigraphsUseNauty()}\]

\[\text{DigraphsUseBliss()}\]

**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 (7.2.15), IsIsomorphicDigraph (7.2.16), IsomorphismDigraphs (7.2.17), and IsomorphismDigraphs (7.2.18) 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 (7.2.17) for more information about isomorphisms of digraphs.

If \( \text{digraph} \) is not a multidigraph then the automorphism group is returned as a group of permutations on the set of vertices of \( \text{digraph} \).

If \( \text{digraph} \) is a multidigraph then the automorphism group is returned as the direct product of a group of permutations on the set of vertices of \( \text{digraph} \) with a group of permutations on the set of edges of \( \text{digraph} \). These groups can be accessed using Projection (Reference: Projection for a domain and a positive integer) on the returned group.

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).

If the argument \( \text{digraph} \) is mutable, then the return value of this attribute is recomputed every time it is called.

```
Example

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

### 7.2.3 BlissAutomorphismGroup (for a digraph)

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

\[ \text{BlissAutomorphismGroup}(\text{digraph}, \text{vertex_colours}) \]

\[ \text{BlissAutomorphismGroup}(\text{digraph}, \text{vertex_colours}, \text{edge_colours}) \]

**Returns:** A permutation group.

If \( \text{digraph} \) is a digraph, then this attribute contains the group of automorphisms of \( \text{digraph} \) 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.
The attribute AutomorphismGroup (7.2.6) returns the value of BlissAutomorphismGroup as
it is not implemented for nauty The requirements for the optional arguments vertex_colours
and edge_colours are documented in AutomorphismGroup (7.2.6). See also DigraphsUseBliss
(7.2.1), and DigraphsUseNauty (7.2.1).
If the argument digraph is mutable, then the return value of this attribute is recomputed every
time it is called.

Example

\begin{verbatim}
gap> G := BlissAutomorphismGroup(JohnsonDigraph(5, 2));;
gap> IsSymmetricGroup(G);
true
gap> Size(G);
120
\end{verbatim}

7.2.4 NautyAutomorphismGroup

NautyAutomorphismGroup(digraph[, vert_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).
If the argument digraph is mutable, then the return value of this attribute is recomputed every
time it is called.

Example

\begin{verbatim}
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) ])
\end{verbatim}

7.2.5 AutomorphismGroup (for a digraph and a homogeneous list)

AutomorphismGroup(digraph, vert_colours)

Returns: A permutation group.

This operation computes the automorphism group of a vertex-coloured digraph. A vertex-coloured
digraph can be specified by its underlying digraph digraph and its colouring vert_colours. Let n
be the number of vertices of digraph. The colouring vert_colours may have one of the following
two forms:

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

The automorphism group of a coloured digraph digraph with colouring vert_colours is
the group consisting of its automorphisms; an automorphism of digraph is an isomorphism
of coloured digraphs from digraph to itself. This group is equal to the subgroup of
AutomorphismGroup\((digraph)\) consisting of those automorphisms that preserve the colouring specified by \(vert\_colours\). See AutomorphismGroup (7.2.2), and see IsomorphismDigraphs (7.2.18) for more information about isomorphisms of coloured digraphs.

If \(digraph\) is not a multidigraph then the automorphism group is returned as a group of permutations on the set of vertices of \(digraph\).

If \(digraph\) is a multidigraph then the automorphism group is returned as the direct product of a group of permutations on the set of vertices of \(digraph\) with a group of permutations on the set of edges of \(digraph\). These groups can be accessed using Projection (Reference: Projection for a domain and a positive integer) on the returned group.

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).

\[\text{gap} > \text{cycle} := \text{CycleDigraph}(9);\]
\[<\text{immutable cycle digraph with 9 vertices}>\]
\[\text{gap} > \text{G} := \text{AutomorphismGroup}(\text{cycle});\]
\[\text{gap} > \text{IsCyclic}(\text{G}) \text{ and Size}(\text{G}) = 9;\]
\[\text{true}\]
\[\text{gap} > \text{colours} := [[1, 4, 7], [2, 5, 8], [3, 6, 9]];\]
\[\text{gap} > \text{H} := \text{AutomorphismGroup}(\text{cycle}, \text{colours});\]
\[\text{gap} > \text{Size}(\text{H});\]
\[3\]
\[\text{gap} > \text{H} = \text{AutomorphismGroup}(\text{cycle}, [1, 2, 3, 1, 2, 3, 1, 2, 3]);\]
\[\text{true}\]
\[\text{gap} > \text{H} = \text{SubgroupByProperty}(\text{G}, \text{p} \rightarrow \text{OnTuplesSets}(\text{colours}, \text{p}) = \text{colours});\]
\[\text{true}\]
\[\text{gap} > \text{IsTrivial}(\text{AutomorphismGroup}(\text{cycle}, [1, 1, 2, 2, 2, 2, 2, 2, 2]));\]
\[\text{true}\]

7.2.6 AutomorphismGroup (for a digraph, homogeneous list, and list)

\[\text{\texttt{AutomorphismGroup}}(\text{digraph}, \textit{vert\_colours}, \textit{edge\_colours})\]

\(\text{(operation)}\)

\(\text{Returns:} \) A permutation group.

This operation computes the automorphism group of a vertex- and/or edge-coloured digraph. A coloured digraph can be specified by its underlying digraph \(\text{digraph}\) and colourings \(\textit{vert\_colours}, \textit{edge\_colours}\). Let \(n\) be the number of vertices of \(\text{digraph}\). The colourings must have the following forms:

- \(\textit{vert\_colours}\) must be \(\text{fail}\) or a list of \(n\) integers, where \(\textit{vert\_colours}[i]\) is the colour of vertex \(i\), using the colours \([1 \ldots m]\) for some \(m \leq n\);

- \(\textit{edge\_colours}\) must be \(\text{fail}\) or a list of \(n\) lists of integers of the same shape as \(\text{OutNeighbours}(\text{digraph})\), where \(\textit{edge\_colours}[i][j]\) is the colour of the edge \(\text{OutNeighbours}(\text{digraph})[i][j]\), using the colours \([1 \ldots k]\) for some \(k \leq n\);

Giving \(\textit{vert\_colours} \ [\textit{edge\_colours}] \) as \(\text{fail}\) is equivalent to setting all vertices [edges] to be the same colour.

Unlike AutomorphismGroup (7.2.2), it is possible to obtain the automorphism group of an edge-coloured multidigraph (see IsMultiDigraph (6.1.10)) when no two edges share the same source,
range, and colour. The automorphism group of a vertex/edge-coloured digraph \textit{digraph} with colouring \(c\) is the group consisting of its vertex/edge-colour preserving automorphisms; an automorphism of \textit{digraph} is an isomorphism of vertex/edge-coloured digraphs from \textit{digraph} to itself. This group is equal to the subgroup of \texttt{AutomorphismGroup} \(\textit{digraph}\) consisting of those automorphisms that preserve the colouring specified by \texttt{colours}. See \texttt{AutomorphismGroup} (7.2.2), and see \texttt{IsomorphismDigraphs} (7.2.18) for more information about isomorphisms of coloured digraphs.

If \textit{digraph} is not a multidigraph then the automorphism group is returned as a group of permutations on the set of vertices of \textit{digraph}.

If \textit{digraph} is a multidigraph then the automorphism group is returned as the direct product of a group of permutations on the set of vertices of \textit{digraph} with a group of permutations on the set of edges of \textit{digraph}. These groups can be accessed using \texttt{Projection} \(\texttt{Projection for a domain and a positive integer}\) on the returned group.

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).

Example

\begin{verbatim}
gap> cycle := CycleDigraph(12); <immutable cycle digraph with 12 vertices>
gap> vert_colours := List([1 .. 12], x -> x mod 3 + 1);;
gap> edge_colours := List([1 .. 12], x -> [x mod 2 + 1]);;
gap> Size(AutomorphismGroup(cycle)); 12
gap> Size(AutomorphismGroup(cycle, vert_colours)); 4
gap> Size(AutomorphismGroup(cycle, fail, edge_colours)); 6
gap> Size(AutomorphismGroup(cycle, vert_colours, edge_colours)); 2
gap> IsTrivial(AutomorphismGroup(cycle, > vert_colours, List([1 .. 12], x -> [x mod 4 + 1]))); true
\end{verbatim}

\subsection{BlissCanonicalLabelling (for a digraph)}

\begin{verbatim}
\texttt{BlissCanonicalLabelling(digraph)} \hspace{1cm} \texttt{NautyCanonicalLabelling(digraph)}
\end{verbatim}

\textbf{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 \(H\):

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

A canonical labelling of a digraph \(G\) (under \(\rho\)) is an isomorphism of \(G\) onto its canonical representative, \(\rho(G)\). See \texttt{IsomorphismDigraphs} (7.2.17) for more information about isomorphisms of digraphs.
BlissCanonicalLabelling returns a canonical labelling of the digraph \textit{digraph} found using \texttt{bliss} by Tommi Junttila and Petteri Kaski. NautyCanonicalLabelling returns a canonical labelling of the digraph \textit{digraph} found using \texttt{nauty} by Brendan McKay and Adolfo Piperno. Note that the canonical labellings returned by \texttt{bliss} and \texttt{nauty} are not usually the same (and may depend on the version used).

BlissCanonicalLabelling can only be computed if \textit{digraph} has no multiple edges; see \texttt{IsMultiDigraph (6.1.10)}.

\begin{verbatim}
Example

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

\subsection{BlissCanonicalLabelling (for a digraph and a list)}

\begin{verbatim}
▷ BlissCanonicalLabelling(digraph, colours)
▷ NautyCanonicalLabelling(digraph, colours)
\end{verbatim}

\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 \(H\):

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

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 (7.2.18)} for more information about isomorphisms of coloured digraphs.

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 \(i\).

If \texttt{digraph} and \texttt{colours} together form a coloured digraph, BlissCanonicalLabelling returns a canonical labelling of the digraph \texttt{digraph} found using \texttt{bliss} by Tommi Junttila and Petteri Kaski. Similarly, NautyCanonicalLabelling returns a canonical labelling of the digraph \texttt{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).

BlissCanonicalLabelling can only be computed if digraph has no multiple edges; see IsMultiDigraph (6.1.10). The canonical labelling of digraph is given as a permutation of its vertices. The canonical representative of digraph can be created from digraph and its canonical labelling p by using the operation OnDigraphs (7.1.1):

```
Example

gap> OnDigraphs(digraph, p);
```

The colouring of the canonical representative can easily be constructed. A vertex v (in 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 digraph. In particular, if colours has the first form that is described above, then the colouring of the canonical representative is given by:

```
Example

gap> List(DigraphVertices(digraph), i -> colours[i / p]);
```

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

```
Example

gap> OnTuplesSets(colours, p);
```

7.2.9 BlissCanonicalDigraph

\begin{itemize}
\item BlissCanonicalDigraph(digraph) (attribute)
\item NautyCanonicalDigraph(digraph) (attribute)
\end{itemize}

**Returns:** A digraph.

The attribute BlissCanonicalLabelling returns the canonical representative found by applying BlissCanonicalLabelling (7.2.7). The digraph returned is canonical in the sense that

- BlissCanonicalDigraph(digraph) and digraph are isomorphic as digraphs; and
If \( gr \) is any digraph then \( \text{BlissCanonicalDigraph}(gr) \) and \( \text{BlissCanonicalDigraph}(\text{digraph}) \) are equal if and only if \( gr \) and \( \text{digraph} \) are isomorphic as digraphs.

Analogously, the attribute \( \text{NautyCanonicalLabelling} \) returns the canonical representative found by applying \( \text{NautyCanonicalLabelling} (7.2.7) \).

If the argument \( \text{digraph} \) is mutable, then the return value of this attribute is recomputed every time it is called.

Example

```gap
digraph := Digraph([[1], [2, 3], [3], [1, 2, 3]]);
<immutable digraph with 4 vertices, 7 edges>
digraph := BlissCanonicalDigraph(digraph);
<immutable digraph with 4 vertices, 7 edges>
OutNeighbours(canon);
[ [ 1 ], [ 2 ], [ 3, 2 ], [ 1, 3, 2 ] ]
```

### 7.2.10 DigraphGroup

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

**Returns:** A permutation group.

If \( \text{digraph} \) is immutable and was created knowing a subgroup of its automorphism group, then this group is stored in the attribute \( \text{DigraphGroup} \). If \( \text{digraph} \) is mutable, or was not created knowing a subgroup of its automorphism group, then \( \text{DigraphGroup} \) returns the entire automorphism group of \( \text{digraph} \). Note that if \( \text{digraph} \) is mutable, then the automorphism group is recomputed every time this function is called.

Note that certain other constructor operations such as \( \text{CayleyDigraph} (3.1.12) \), \( \text{BipartiteDoubleDigraph} (3.3.37) \), and \( \text{DoubleDigraph} (3.3.36) \), may not require a group as one of the arguments, but use the standard constructor method using a group, and hence set the \( \text{DigraphGroup} \) attribute for the resulting digraph.

Example

```gap
n := 4;;
dig := function(x, y)
> return ((x - y) mod n) = 1 or (((x - y) mod n) = n - 1);
> end;;
group := CyclicGroup(IsPermGroup, n);
Group([ (1,2,3,4) ])
digraph := Digraph(IsMutableDigraph, group, [1 .. n], \^, dig);
<mutable digraph with 4 vertices, 8 edges>
HasDigraphGroup(digraph);
false
digraph := Digraph(IsImmutableDigraph, group, [1 .. n], \^, dig);
Group([ (2,4), (1,2)(3,4) ])
AutomorphismGroup(digraph);
Group([ (2,4), (1,2)(3,4) ])
digraph := Digraph(group, [1 .. n], \^, dig);
<immutable digraph with 4 vertices, 8 edges>
HasDigraphGroup(digraph);
true
digraph := Digraph(group, [1 .. n], \^, dig);
Group([ (1,2,3,4) ])
digraph := DoubleDigraph(digraph);
```
Digraphs

7.2.11 DigraphOrbits

\(\text{DigraphOrbits} \) (attribute)

\textbf{Returns:} An immutable list of lists of integers.

DigraphOrbits returns the orbits of the action of the DigraphGroup (7.2.10) on the set of vertices of \(\text{digraph} \).

Example

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

7.2.12 DigraphOrbitReps

\(\text{DigraphOrbitReps} \) (attribute)

\textbf{Returns:} An immutable list of integers.

DigraphOrbitReps returns a list of orbit representatives of the action of the DigraphGroup (7.2.10) on the set of vertices of \(\text{digraph} \).
Example

```gap
gap> D := CayleyDigraph(AlternatingGroup(4));
<immutable digraph with 12 vertices, 24 edges>
gap> DigraphOrbitReps(D);
[ 1 ]
gap> D := DigraphMutableCopy(D);
<mutable digraph with 12 vertices, 24 edges>
gap> DigraphOrbitReps(D);
[ 1 ]
gap> D := DigraphFromDigraph6String("&IGO??S?¢?_@?a?CK?O");
<immutable digraph with 10 vertices, 14 edges>
gap> DigraphOrbitReps(D);
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
gap> DigraphOrbitReps(DigraphMutableCopy(D));
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

### 7.2.13 DigraphSchreierVector

*DigraphSchreierVector*(digraph)  

**Returns:** An immutable list of integers.

DigraphSchreierVector returns the so-called Schreier vector of the action of the DigraphGroup (7.2.10) on the set of vertices of digraph. The Schreier vector is a list sch of integers with length DigraphNrVertices(digraph) where:

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

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

Example

```gap
gap> n := 4;;
gap> adj := function(x, y)
   > return (((x - y) mod n) = 1) or (((x - y) mod n) = n - 1);
   > end;
> end;
gap> group := CyclicGroup(IsPermGroup, n);
Group([ (1,2,3,4) ])
gap> D := Digraph(IsMutableDigraph, group, [1 .. n], ^, adj);
<mutable digraph with 4 vertices, 8 edges>
gap> sch := DigraphSchreierVector(D);
[ -1, 2, 2, 1 ]
gap> D := CayleyDigraph(AlternatingGroup(4));
<immutable digraph with 12 vertices, 24 edges>
gap> sch := DigraphSchreierVector(D);
[ -1, 2, 2, 1, 1, 1, 1, 1, 2, 2, 2, 1 ]
gap> DigraphOrbitReps(D);
[ 1 ]
gap> gens := GeneratorsOfGroup(DigraphGroup(D));
[ (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
```
7.2.14 DigraphStabilizer

\[ \text{DigraphStabilizer}(\text{digraph}, v) \]

| Returns: | A permutation group. |

DigraphStabilizer returns the stabilizer of the vertex \( v \) under of the action of the DigraphGroup (7.2.10) on the set of vertices of \( \text{digraph} \).

Example

\[
\begin{align*}
gap> D := \text{DigraphFromDigraph6String}(&GYHPQgWTIIPW); \\
<\text{immutable digraph with 8 vertices, 24 edges}> \\
gap> \text{DigraphStabilizer}(D, 8); \\
\text{Group}(()); \\
gap> \text{DigraphStabilizer}(D, 2); \\
\text{Group}(()); \\
gap> D := \text{DigraphMutableCopy}(D); \\
<\text{mutable digraph with 8 vertices, 24 edges}> \\
gap> \text{DigraphStabilizer}(D, 8); \\
\text{Group}(()); \\
gap> \text{DigraphStabilizer}(D, 2); \\
\text{Group}(());
\end{align*}
\]

7.2.15 IsIsomorphicDigraph (for digraphs)

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

| Returns: | true or false. |

This operation returns true if there exists an isomorphism from the digraph \( \text{digraph1} \) to the digraph \( \text{digraph2} \). See IsomorphismDigraphs (7.2.17) 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).

Example

\[
\begin{align*}
gap> \text{digraph1} := \text{CycleDigraph}(4); \\
<\text{immutable cycle digraph with 4 vertices}> \\
gap> \text{digraph2} := \text{CycleDigraph}(5); \\
<\text{immutable cycle digraph with 5 vertices}> \\
gap> \text{IsIsomorphicDigraph}(\text{digraph1}, \text{digraph2}); \\
\text{false} \\
gap> \text{digraph2} := \text{DigraphReverse}(\text{digraph1}); \\
<\text{immutable digraph with 4 vertices, 4 edges}> \\
gap> \text{IsIsomorphicDigraph}(\text{digraph1}, \text{digraph2}); \\
\text{true} \\
gap> \text{digraph1} := \text{Digraph}([[3], [], []]); \\
<\text{immutable digraph with 3 vertices, 1 edge}> \\
gap> \text{digraph2} := \text{Digraph}([[], [], [2]]);
\end{align*}
\]
7.2.16  

**IsIsomorphicDigraph (for digraphs and homogeneous lists)**

\[ \text{IsIsomorphicDigraph}(\text{digraph1}, \text{digraph2}, \text{colours1}, \text{colours2}) \]

(operation)

**Returns:** true or false.

This operation tests for isomorphism of coloured digraphs. A coloured digraph can be specified by its underlying digraph \( \text{digraph1} \) and its colouring \( \text{colours1} \). Let \( n \) be the number of vertices of \( \text{digraph1} \). The colouring \( \text{colours1} \) 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 \( \text{DigraphVertices} (\text{digraph}) \), such that \( \text{colours}[i] \) is the list of all vertices with colour \( i \).

If \( \text{digraph1} \) and \( \text{digraph2} \) are digraphs without multiple edges, and \( \text{colours1} \) and \( \text{colours2} \) are colourings of \( \text{digraph1} \) and \( \text{digraph2} \), respectively, then this operation returns true if there exists an isomorphism between these two coloured digraphs. See **IsomorphismDigraphs** (7.2.18) for more information about isomorphisms of coloured digraphs.

By default, an isomorphism is found using the canonical labellings of the digraphs obtained from 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 **DigraphsUseBliss** (7.2.1), and **DigraphsUseNauty** (7.2.1).

Example

\[
\text{gap> digraph1} := \text{ChainDigraph}(4); \\
\text{<immutable chain digraph with 4 vertices>} \\
\text{gap> digraph2} := \text{ChainDigraph}(3); \\
\text{<immutable chain digraph with 3 vertices>} \\
\text{gap> IsIsomorphicDigraph(digraph1, digraph2,} \\
\text{> \quad \text{[[[1, 4], [2, 3]], [[1, 2], [3]]];} \\
\text{false} \\
\text{gap> digraph2} := \text{DigraphReverse(digraph1)}; \\
\text{<immutable digraph with 4 vertices, 3 edges>} \\
\text{gap> IsIsomorphicDigraph(digraph1, digraph2,} \\
\text{> \quad [1, 1, 1, 1], [1, 1, 1, 1];} \\
\text{true} \\
\text{gap> IsIsomorphicDigraph(digraph1, digraph2,} \\
\text{> \quad [1, 2, 2, 1], [1, 2, 2, 1];} \\
\text{true} \\
\text{gap> IsIsomorphicDigraph(digraph1, digraph2,} \\
\text{> \quad [1, 1, 2, 2], [1, 1, 2, 2];} \\
\text{false}
\]

7.2.17  

**IsomorphismDigraphs (for digraphs)**

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

(operation)

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

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

```
gap> digraph1 := CycleDigraph(4);
<immutable cycle digraph with 4 vertices>
gap> digraph2 := CycleDigraph(5);
<immutable cycle digraph with 5 vertices>
gap> IsomorphismDigraphs(digraph1, digraph2);
fail

gap> digraph1 := CompleteBipartiteDigraph(10, 5);
<immutable complete bipartite digraph with bicomponent sizes 10 and 5>
gap> digraph2 := CompleteBipartiteDigraph(5, 10);
<immutable complete bipartite digraph with bicomponent sizes 5 and 10>
gap> p := IsomorphismDigraphs(digraph1, digraph2);
(1,6,11)(2,7,12)(3,8,13)(4,9,14)(5,10,15)

gap> OnDigraphs(digraph1, p) = digraph2;
true
```

### Example

#### 7.2.18 IsomorphismDigraphs (for digraphs and homogeneous lists)

\(\triangleright\) \texttt{IsomorphismDigraphs(digraph1, digraph2, colours1, colours2)} (operation)

**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 \(n\) 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 \([1 .. m]\) for some \(m \leq n\); or

- a list of non-empty disjoint lists whose union is \texttt{DigraphVertices(digraph1)}, such that \texttt{colours[i]} is the list of all vertices with colour \(i\).

An isomorphism between coloured digraphs is an isomorphism between the underlying digraphs that preserves the colourings. See \texttt{IsomorphismDigraphs (7.2.17)} for more information about isomorphisms of digraphs. More precisely, let \(f\) 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 \(p\) be the permutation of the vertices of \texttt{digraph1} that corresponds to \(f\). Then \(f\) preserves the colourings of \texttt{digraph1} and \texttt{digraph2} – and hence is an isomorphism of coloured digraphs – if \texttt{colours1[i]} = \texttt{colours2[i \sim p]} for all vertices \(i\) 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 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).

Example

```
gap> digraph1 := ChainDigraph(4);
<immutable chain digraph with 4 vertices>
gap> digraph2 := ChainDigraph(3);
<immutable chain digraph with 3 vertices>
gap> IsomorphismDigraphs(digraph1, digraph2,
>    [[1, 4], [2, 3]], [[1, 2], [3]]);
fail
```

```
gap> digraph2 := DigraphReverse(digraph1);
<immutable digraph with 4 vertices, 3 edges>
gap> colours1 := [1, 1, 1, 1];;
gap> colours2 := [1, 1, 1, 1];;
gap> p := IsomorphismDigraphs(digraph1, digraph2, colours1, colours2);
(1,4)(2,3)
gap> OnDigraphs(digraph1, p) = digraph2;
true
```

```
gap> List(DigraphVertices(digraph1), i -> colours1[i ^ p]) = colours2;
true
```

```
gap> colours1 := [1, 1, 2, 2];;
gap> colours2 := [2, 2, 1, 1];;
gap> p := IsomorphismDigraphs(digraph1, digraph2, colours1, colours2);
(1,4)(2,3)
gap> OnDigraphs(digraph1, p) = digraph2;
true
```

```
gap> List(DigraphVertices(digraph1), i -> colours1[i ^ p]) = colours2;
true
```

```
gap> IsomorphismDigraphs(digraph1, digraph2,
>    [1, 1, 2, 2], [1, 1, 2, 2]);
fail
```

7.2.19 RepresentativeOutNeighbours

\> RepresentativeOutNeighbours(digraph) (attribute)

Returns: An immutable list of lists.

This function returns the list out of out-neighbours of each representative of the orbits of the action of DigraphGroup (7.2.10) 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 (7.2.10) is trivial, then OutNeighbours (5.2.6) is returned.

Example

```
gap> D := Digraph([  
>  [2, 1, 3, 4, 5], [3, 5], [2], [1, 2, 3, 5], [1, 2, 3, 4]]);
<immutable digraph with 5 vertices, 16 edges>
gap> DigraphGroup(D);
Group(())
gap> RepresentativeOutNeighbours(D);
[ [ 2, 1, 3, 4, 5 ], [ 3, 5 ], [ 2 ], [ 1, 2, 3, 5 ], [ 1, 2, 3, 4 ] ]
```
Digraphs

gap> D := Digraph(IsMutableDigraph, [
> [2, 1, 3, 4, 5], [3, 5], [2], [1, 2, 3, 5], [1, 2, 3, 4]]);
<mutable digraph with 5 vertices, 16 edges>
gap> DigraphGroup(D);
Group(())
gap> RepresentativeOutNeighbours(D);
[ [ 2, 1, 3, 4, 5 ], [ 3, 5 ], [ 2 ], [ 1, 2, 3, 5 ], [ 1, 2, 3, 4 ] ]
gap> D := DigraphFromDigraph6String("&GYHPQgWTIIPW");
<immutable digraph with 8 vertices, 24 edges>
gap> G := DigraphGroup(D);
gap> GeneratorsOfGroup(G);
[ (1,2)(3,4)(5,6)(7,8), (1,3,2,4)(5,7,6,8), (1,5)(2,6)(3,8)(4,7) ]
gap> Set(RepresentativeOutNeighbours(D), Set);
[ [ 2, 3, 5 ] ]

7.2.20 IsDigraphIsomorphism (for digraphs and transformation or permutation)

\[ \text{IsDigraphIsomorphism(src, ran, x)} \]
\[ \text{IsDigraphIsomorphism(src, ran, x, col1, col2)} \]
\[ \text{IsDigraphAutomorphism(digraph, x)} \]
\[ \text{IsDigraphAutomorphism(digraph, x, col)} \]

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).

If \(col1\) and \(col2\), or \(col\), are given, then they must represent vertex colourings; see AutomorphismGroup (7.2.5) for details of the permissible values for these arguments. The homomorphism must then also have the property:

- \(col1[i] = col2[i \sim x]\) for all vertices \(i\) of \(src\), for IsDigraphIsomorphism.
- \(col[i] = col[i \sim x]\) for all vertices \(i\) of \(digraph\), for IsDigraphAutomorphism.

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

Example

\[ \text{gap> src := Digraph([[1], [1, 2], [1, 3]]); } \]
\(<\text{immutable digraph with 3 vertices, 5 edges}>\]
\[ \text{gap> IsDigraphAutomorphism(src, (1, 2, 3));} \]
false
7.2.21 IsDigraphColouring

⇒ IsDigraphColouring(digraph, list)  (operation)
⇒ IsDigraphColouring(digraph, t)  (operation)

**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);
<immutable symmetric 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
```
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

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

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 \( D2 \) in aut_grp.

**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 the function returns true, then HomomorphismDigraphsFinder stops and does not find any further homomorphisms. This feature might be useful if you are searching for a homomorphism that satisfies some condition that you cannot specify via the other arguments to 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 \( \text{hint} \) is a positive integer, then only homomorphisms of rank \( \text{hint} \) are found.

If \( \text{hint} \) is \( \text{fail} \), 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, \( \text{injective} \) can also be \( \text{false} \) or \( \text{true} \) 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 \( D_2 \). \( \text{HomomorphismDigraphsFinder} \) only finds homomorphisms from \( D_1 \) to the subgraph of \( D_2 \) induced by the vertices \( \text{image} \).

**partial_map**
- This argument should be a partial map from \( D_1 \) to \( D_2 \), that is, a (not necessarily dense) list of vertices of the digraph \( D_2 \) of length no greater than the number vertices in the digraph \( D_1 \). \( \text{HomomorphismDigraphsFinder} \) only finds homomorphisms extending \( \text{partial_map} \) (if any).

**colors1**
- This should be a list representing possible colours of vertices in the digraph \( D_1 \); see \( \text{AutomorphismGroup} \) (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 \( D_2 \); see \( \text{AutomorphismGroup} \) (7.2.5) for details of the permissible values for this argument.

**order**
- The optional argument \( \text{order} \) specifies the order the vertices in \( D_1 \) appear in the search for 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 \( \text{DigraphWelshPowellOrder} \) (7.3.16), i.e. vertices ordered from highest to lowest degree. The optional argument \( \text{aut_grp} \) should be a subgroup of the automorphism group of \( D_2 \). This function returns unique representatives of the homomorphisms found up to right multiplication by \( \text{aut_grp} \). If this argument is not specific, it defaults to the full automorphism group of \( D_2 \), which may be costly to calculate.

```gap
gap> D := ChainDigraph(10);
<immutable chain digraph with 10 vertices>
gap> D := DigraphSymmetricClosure(D);
<immutable symmetric 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 := CompleteDigraph(6);
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)
```
7.3.2 DigraphHomomorphism

\textbf{DigraphHomomorphism}(\textit{digraph1}, \textit{digraph2}) \quad (\text{operation})

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

A homomorphism from \textit{digraph1} to \textit{digraph2} is a mapping from the vertex set of \textit{digraph1} to a subset of the vertices of \textit{digraph2}, such that every pair of vertices \([i,j]\) which has an edge \(i \rightarrow j\) is mapped to a pair of vertices \([a,b]\) which has an edge \(a \rightarrow b\). Note that non-adjacent vertices can still be mapped to adjacent vertices.

\text{DigraphHomomorphism} returns a single homomorphism between \textit{digraph1} and \textit{digraph2} if it exists, otherwise it returns \texttt{fail}.

\textbf{Example}

\begin{verbatim}
    gap> gr1 := ChainDigraph(3);;
    <immutable digraph with 3 vertices, 2 edges>
    gap> gr2 := Digraph([[3, 5], [2], [3, 1], [], [4]]);
    <immutable digraph with 5 vertices, 6 edges>
    gap> DigraphHomomorphism(gr1, gr2);
    IdentityTransformation
    gap> map := DigraphHomomorphism(gr1, gr2);
    Transformation( [ 3, 1, 5, 4, 5 ]
    gap> IsDigraphHomomorphism(gr1, gr2, map);
    true
\end{verbatim}

7.3.3 HomomorphismsDigraphs

\textbf{HomomorphismsDigraphs} (\textit{digraph1}, \textit{digraph2}) \quad (\text{operation})

\textbf{HomomorphismsDigraphsRepresentatives} (\textit{digraph1}, \textit{digraph2}) \quad (\text{operation})

\textbf{Returns:} A list of transformations.

\text{HomomorphismsDigraphsRepresentatives} finds every \text{DigraphHomomorphism} (7.3.2) between \textit{digraph1} and \textit{digraph2}, up to right multiplication by an element of the \text{AutomorphismGroup} (7.2.2) of \textit{digraph2}. In other words, every homomorphism \(f\) between \textit{digraph1}
and digraph2 can be written as the composition \( f = g \circ x \), where \( g \) is one of the
HomomorphismsDigraphsRepresentatives and \( x \) is an automorphism of digraph2.

HomomorphismsDigraphs returns all homomorphisms between digraph1 and digraph2.

Example:

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

7.3.4 DigraphMonomorphism

\textsc{DigraphMonomorphism}(\textit{digraph1, digraph2}) (operation)

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

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

Example:

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

\textsc{MonomorphismsDigraphs}(\textit{digraph1, digraph2}) (operation)

\textsc{MonomorphismsDigraphsRepresentatives}(\textit{digraph1, digraph2}) (operation)

\textbf{Returns:} A list of transformations.

These operations behave the same as HomomorphismsDigraphs (7.3.3) and
HomomorphismsDigraphsRepresentatives (7.3.3), except they only return \textit{injective} homo-
morphisms.

Example:

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

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

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

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

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

gap> DigraphEpimorphism(gr1, gr2);
Transformation( [ 3, 1, 2, 3 ] )
\end{verbatim}

7.3.7 EpimorphismsDigraphs

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

\textbf{Returns:} A list of transformations.

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

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

7.3.8 DigraphEmbedding

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

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

An embedding of a digraph \text{digraph1} into another digraph \text{digraph2} is a DigraphMonomorphism (7.3.4) from \text{digraph1} to \text{digraph2} which has the additional prop-
erty that a pair of vertices \([i, j]\) which have no edge \(i \rightarrow j\) in digraph1 are mapped to a pair of vertices \([a, b]\) which have no edge \(a \rightarrow b\) in digraph2.

In other words, an embedding \(t\) is an isomorphism from digraph1 to the InducedSubdigraph (3.3.3) of digraph2 on the image of \(t\).

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

Example

\[
\text{gap> gr := ChainDigraph(3);}
\text{<immutable chain digraph with 3 vertices>}
\text{gap> DigraphEmbedding(gr, CompleteDigraph(4));}
\text{fail}
\text{gap> DigraphEmbedding(gr, Digraph([[3], [1, 4], [1], [3]])};}
\text{Transformation( [ 2, 4, 3, 4 ] )}
\]

### 7.3.9 EmbeddingsDigraphs

- `EmbeddingsDigraphs(D1, D2)`
- `EmbeddingsDigraphsRepresentatives(D1, D2)`

**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

\[
\text{gap> D1 := NullDigraph(2);}
\text{<immutable empty digraph with 2 vertices>}
\text{gap> D2 := CycleDigraph(5);}
\text{<immutable cycle digraph with 5 vertices>}
\text{gap> EmbeddingsDigraphsRepresentatives(D1, D2);}
\text{[ Transformation( [ 1, 3, 3 ] ), Transformation( [ 1, 4, 3, 4 ] ) ]}
\text{gap> EmbeddingsDigraphs(D1, D2);} \\
\text{[ Transformation( [ 1, 3, 3 ] ), Transformation( [ 1, 4, 3, 4 ] ),}
\text{ Transformation( [ 2, 4, 4, 5, 1 ] ),}
\text{ Transformation( [ 2, 5, 4, 5, 1 ] ),}
\text{ Transformation( [ 3, 1, 5, 1, 2 ] ),}
\text{ Transformation( [ 3, 5, 5, 1, 2 ] ),}
\text{ Transformation( [ 4, 1, 1, 2, 3 ] ),}
\text{ Transformation( [ 4, 2, 1, 2, 3 ] ),}
\text{ Transformation( [ 5, 2, 2, 3, 4 ] ),}
\text{ Transformation( [ 5, 3, 2, 3, 4 ] ) ]}
\]

### 7.3.10 IsDigraphHomomorphism (for digraphs and a permutation or transformation)

- `IsDigraphHomomorphism(src, ran, x)`
- `IsDigraphHomomorphism(src, ran, x, col1, col2)`
- `IsDigraphEpimorphism(src, ran, x)`
- `IsDigraphEpimorphism(src, ran, x, col1, col2)`
- `IsDigraphMonomorphism(src, ran, x)`
- `IsDigraphMonomorphism(src, ran, x, col1, col2)`
- `IsDigraphEndomorphism(digraph, x)`
\textbf{IsDigraphEndomorphism}(\textit{digraph}, \textit{x}, \textit{col})

\textbf{Returns}: true or false.

\textit{IsDigraphEndomorphism} returns true if the permutation or transformation \( x \) is an endomorphism from the digraph \( \text{src} \) to the digraph \( \text{ran} \).

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

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

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

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

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

See also \texttt{GeneratorsOfEndomorphismMonoid}(7.3.13).

If \textit{col1} and \textit{col2}, or \textit{col}, are given, then they must represent vertex colourings; see \texttt{AutomorphismGroup}(7.2.5) for details of the permissible values for these argument. The homomorphism must then also have the property:

- \( \text{col}[i] = \text{col}[i \mapsto x] \) for all vertices \( i \) of \( \text{digraph} \), in the case of \textit{IsDigraphEndomorphism}.

- \( \text{col1}[i] = \text{col2}[i \mapsto x] \) for all vertices \( i \) of \( \text{src} \), in the cases of the other operations.

See also \texttt{DigraphsRespectsColouring}(7.3.12).

Example

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

\[
gap> \text{IsDigraphEpimorphism(src, ran, Transformation([3, 3, 3]))};
\]
false

\[
gap> \text{IsDigraphMonomorphism(src, ran, Transformation([1, 2, 2]))};
\]
false

\[
gap> \text{IsDigraphEpimorphism(src, ran, Transformation([1, 2, 2]))};
\]
true

\[
gap> \text{IsDigraphMonomorphism(ran, src, ()));}
\]
true

### 7.3.11 IsDigraphEmbedding (for digraphs and a permutation or transformation)

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

\[
\text{IsDigraphEmbedding(src, ran, x, col1, col2)}
\]

**Returns:** true or false.

IsDigraphEmbedding returns true if the permutation or transformation \(x\) is an embedding of the digraph \(src\) into the digraph \(ran\), while respecting the colourings \(col1\) and \(col2\) if given.

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

#### Example

\[
\text{gap> src := Digraph([[1], [1, 2]]);}
\]
<immutable digraph with 2 vertices, 3 edges>

\[
\text{gap> ran := Digraph([[1, 1, 1], [1, 2], [1, 3]]);}
\]
<immutable digraph with 3 vertices, 5 edges>

\[
\text{gap> IsDigraphMonomorphism(src, ran, ());}
\]
true

\[
\text{gap> IsDigraphEmbedding(src, ran, ());}
\]
true

\[
\text{gap> IsDigraphEmbedding(src, ran, (), [2, 1], [2, 1, 1]);}
\]
true

\[
\text{gap> IsDigraphEmbedding(src, ran, (), [2, 1], [1, 2, 1]);}
\]
false

\[
\text{gap> ran := Digraph([[1, 2], [1, 2], [1, 3]]);}
\]
<immutable digraph with 3 vertices, 6 edges>

\[
\text{gap> IsDigraphMonomorphism(src, ran, IdentityTransformation);}
\]
true

\[
\text{gap> IsDigraphEmbedding(src, ran, IdentityTransformation);}
\]
false

### 7.3.12 DigraphsRespectsColouring

\[
\text{DigraphsRespectsColouring(src, ran, x, col1, col2)}
\]

**Returns:** true or false.

The operation DigraphsRespectsColouring verifies whether or not the permutation or transformation \(x\) respects the vertex colourings \(col1\) and \(col2\) of the digraphs \(src\) and \(range\). That is, DigraphsRespectsColouring returns true if and only if for all vertices \(i\) of \(src\), \(col1[i] = col2[i^{-x}]\).
Example

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

### 7.3.13 GeneratorsOfEndomorphismMonoid

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

**Returns:** A list of transformations.

An endomorphism of `digraph` is a homomorphism `DigraphHomomorphism` (7.3.2) from `digraph` back to itself. `GeneratorsOfEndomorphismMonoid`, called with a single argument, returns a generating set for the monoid of all endomorphisms of `digraph`. If `digraph` belongs to `IsImmutableDigraph` (3.1.3), then the value of `GeneratorsOfEndomorphismMonoid` will not be recomputed on future calls.

If the `colors` argument is specified, then `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:

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

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

Example

```gap
gap> gr := Digraph(List([1 .. 3], x -> [1 .. 3]));;
gap> GeneratorsOfEndomorphismMonoid(gr);
[ Transformation( [ 1, 3, 2 ] ), Transformation( [ 2, 1 ] ),
  IdentityTransformation, Transformation( [ 1, 2, 1 ] ),
  Transformation( [ 1, 2, 2 ] ), Transformation( [ 1, 1, 2 ] ),
  Transformation( [ 1, 1, 1 ] ) ]
gap> GeneratorsOfEndomorphismMonoid(gr, 3);
[ Transformation( [ 1, 3, 2 ] ), Transformation( [ 2, 1 ] ),
  IdentityTransformation ]
gap> gr := CompleteDigraph(3);;
gap> GeneratorsOfEndomorphismMonoid(gr);
[ Transformation( [ 2, 3, 1 ] ), Transformation( [ 2, 1 ] ),
  IdentityTransformation ]
gap> GeneratorsOfEndomorphismMonoid(gr, [1, 2, 2]);
[ Transformation( [ 1, 3, 2 ] ), IdentityTransformation ]
gap> GeneratorsOfEndomorphismMonoid(gr, [[1], [2, 3]]);
[ Transformation( [ 1, 3, 2 ] ), IdentityTransformation ]
gap> GeneratorsOfEndomorphismMonoid(gr, [1, 2, 2], 2);
[ Transformation( [ 1, 3, 2 ] ), Transformation( [ 2, 1 ] ),
  IdentityTransformation ]
gap> GeneratorsOfEndomorphismMonoid(gr, [[1], [2, 3]], 2);
[ Transformation( [ 1, 3, 2 ] ), Transformation( [ 2, 1 ] ),
  IdentityTransformation ]
```
7.3.14 DigraphColouring (for a digraph and a number of colours)

\[
\text{DigraphColouring(digraph, n)}
\]

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 (7.3.15) 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

\[
\begin{align*}
gap & \text{DigraphColouring(CompleteDigraph(5), 4);} \\
& \text{fail} \\
gap & \text{DigraphColouring(ChainDigraph(10), 1);} \\
& \text{fail} \\
gap & \text{D := ChainDigraph(10);} \\
gap & \text{t := DigraphColouring(D, 2);} \\
& \text{Transformation( [ 1, 2, 1, 2, 1, 2, 1, 2, 1, 2 ] )} \\
gap & \text{IsDigraphColouring(D, t);} \\
& \text{true} \\
gap & \text{DigraphGreedyColouring(D);} \\
& \text{Transformation( [ 2, 1, 2, 1, 2, 1, 2, 1, 2, 1 ] )}
\end{align*}
\]

7.3.15 DigraphGreedyColouring (for a digraph and vertex order)

\[
\text{DigraphGreedyColouring(digraph, order)} \\
\text{DigraphGreedyColouring(digraph, func)} \\
\text{DigraphGreedyColouring(digraph)}
\]

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 (7.3.16) is used by default.

See also DigraphColouring (7.3.14).

Example

\[
\begin{align*}
gap & \text{DigraphGreedyColouring(ChainDigraph(10));} \\
& \text{Transformation( [ 2, 1, 2, 1, 2, 1, 2, 1, 2, 1 ] )}
\end{align*}
\]
7.3.16 DigraphWelshPowellOrder

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

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

DigraphWelshPowellOrder returns a list of all of the vertices of the digraph \( \text{digraph} \) ordered according to the sum of the number of out- and in-neighbours, from highest to lowest.

Example

\[
\text{gap> \hspace{1cm} DigraphWelshPowellOrder(Digraph([4, [9], [9], [],
\hspace{1cm} [4, 6, 9], [1], [], [],
\hspace{1cm} [4, 5], [4, 5]]));
\]

\[
[5, 9, 4, 1, 6, 10, 2, 3, 7, 8]
\]

7.3.17 ChromaticNumber

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

**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 \( \text{digraph} \) 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 \( \text{digraph} \) with \( n \) colours.

In other words, for a digraph with at least one vertex, ChromaticNumber returns the least number \( n \) such that DigraphColouring(\( \text{digraph} \), \( n \)) does not return fail. See DigraphColouring (7.3.14).

Example

\[
\text{gap> \hspace{1cm} ChromaticNumber(NullDigraph(10));
\hspace{1cm} 1 \]
\]
\[
\text{gap> \hspace{1cm} ChromaticNumber(CompleteDigraph(10));
\hspace{1cm} 10 \]
\]
\[
\text{gap> \hspace{1cm} ChromaticNumber(CompleteBipartiteDigraph(5, 5));
\hspace{1cm} 2 \]
\]
\[
\text{gap> \hspace{1cm} ChromaticNumber(Digraph([[], [3], [5], [2, 3], [4]]));
\hspace{1cm} 3 \]
\]
\[
\text{gap> \hspace{1cm} ChromaticNumber(NullDigraph(0));
\hspace{1cm} 0 \]
\]
\[
\text{gap> \hspace{1cm} D := PetersenGraph(IsMutableDigraph);
\hspace{1cm} <\text{mutable digraph with 10 vertices, 30 edges}>
\]
\[
\text{gap> \hspace{1cm} ChromaticNumber(D);}
\hspace{1cm} 3 \]
\]

7.3.18 DigraphCore

\[\text{DigraphCore}(\text{D})\]

**Returns:** A list of positive integers.

If \( D \) is a digraph, then DigraphCore returns a list of vertices corresponding to the core of \( D \). In particular, the subdigraph of \( D \) induced by this list is isomorphic to the core of \( D \).
The core of a digraph $D$ is the minimal subdigraph $C$ of $D$ which is a homomorphic image of $D$. The core of a digraph is unique up to isomorphism.

Example

\begin{verbatim}
gap> D := DigraphSymmetricClosure(CycleDigraph(8));
<immutable symmetric digraph with 8 vertices, 16 edges>
gap> DigraphCore(D);
[ 1, 2 ]
gap> D := PetersenGraph();
<immutable digraph with 10 vertices, 30 edges>
gap> DigraphCore(D);
[ 1 .. 10 ]
gap> D := Digraph(IsMutableDigraph, [[3], [3], [4], [5], [2]]);
<mutable digraph with 5 vertices, 5 edges>
gap> DigraphCore(D);
[ 2, 3, 4, 5 ]
\end{verbatim}
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 lengthy, 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 (6.1.10) 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 (3.3.24) and DigraphRemoveAllMultipleEdges (3.3.25) 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 (3.3.5) for more information about this. 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 (3.3.11). 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 (3.3.10), 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) ▶️
  - 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 that is not properly contained in another clique. A clique is permitted, but not required, to contain vertices at which there is a loop.

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

\[ \text{CliquesFinder}(\text{digraph}, \text{hook}, \text{user\_param}, \text{limit}, \text{include}, \text{exclude}, \text{max}, \text{size}, \text{reps}) \]

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

This function finds cliques of the digraph \text{digraph} subject to the conditions imposed by the other arguments as described below. Note that a clique is represented by the immutable list of the vertices that it contains.

Let \( G \) denote the automorphism group of the maximal symmetric subdigraph of \text{digraph} without loops (see AutomorphismGroup (7.2.2) and MaximalSymmetricSubdigraphWithoutLoops (3.3.5)).

\textbf{hook}

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

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

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

\textbf{user\_param}

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

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

\textbf{limit}

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

\textbf{include and exclude}

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

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

\textbf{max}

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

\textbf{size}

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

\textbf{reps}

This argument should be \text{true} or \text{false}.
If `reps` is true then the arguments `include` and `exclude` are each required to be invariant under the action of $G$ on sets of vertices. In this case, `CliquesFinder` will find representatives of the orbits of the desired cliques under the action of $G$, although representatives may be returned that are in the same orbit. If `reps` is false then `CliquesFinder` 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.

```gap
Examples

gap> D := CompleteDigraph(5);
<immutable complete digraph with 5 vertices>
gap> user_param := [];;
gap> f := function(a, b) # Calculate size of clique
> AddSet(user_param, Size(b));
> end;;
gap> CliquesFinder(D, f, user_param, infinity, [], [], false, fail,
> true);
[ 1, 2, 3, 4, 5 ]
gap> CliquesFinder(D, f, [], 5, [2, 4], [3], false, fail, false);
[ [ 2, 4 ], [ 1, 2, 4 ], [ 2, 4, 5 ], [ 1, 2, 4, 5 ] ]
gap> CliquesFinder(D, f, [], 2, [2, 4], [3], false, fail, false);
[ [ 2, 4 ], [ 1, 2, 4 ] ]
gap> CliquesFinder(D, f, [], infinity, [], [], true, 5, false);
[ 1, 2, 3, 4, 5 ]
gap> CliquesFinder(D, f, [], infinity, [1, 3], [], false, 3, false);
[ [ 1, 2, 3 ], [ 1, 3, 4 ], [ 1, 3, 5 ] ]
gap> CliquesFinder(D, f, [], infinity, [1, 3], [], true, 3, false);
[ ]
gap> D := CompleteDigraph(IsMutableDigraph, 5);
<mutable 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> CliquesFinder(D, f, user_param, infinity, [], [], false, fail,
> true);
[ 1, 2, 3, 4, 5 ]
gap> CliquesFinder(D, f, [], 5, [2, 4], [3], false, fail, false);
[ [ 2, 4 ], [ 1, 2, 4 ], [ 2, 4, 5 ], [ 1, 2, 4, 5 ] ]
gap> CliquesFinder(D, f, [], 2, [2, 4], [3], false, fail, false);
[ [ 2, 4 ], [ 1, 2, 4 ] ]
gap> CliquesFinder(D, f, [], infinity, [], [], true, 5, false);
[ 1, 2, 3, 4, 5 ]
gap> CliquesFinder(D, f, [], infinity, [1, 3], [], false, 3, false);
[ [ 1, 2, 3 ], [ 1, 3, 4 ], [ 1, 3, 5 ] ]
gap> CliquesFinder(D, f, [], infinity, [1, 3], [], true, 3, false);
[ ]
```
8.1.3 DigraphClique

\[ \text{DigraphClique} (\text{digraph}, \text{include}, \text{exclude}, \text{size}) \] (function)
\[ \text{DigraphMaximalClique} (\text{digraph}, \text{include}, \text{exclude}, \text{size}) \] (function)

**Returns:** An immutable list of positive integers, or fail.

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

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

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

**One or two arguments**

If one or two arguments are supplied, then \text{DigraphClique} and \text{DigraphMaximalClique} greedily enlarge the clique \text{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 \text{DigraphMaximalCliques} (8.1.4) with a limit of 1.

**Three arguments**

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

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

**Four arguments**

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

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

---

**Example**

```gap
gap> D := Digraph([[2, 3, 4], [1, 3], [1, 2], [1, 5], []]);
gap> IsSymmetricDigraph(D);
false
gap> DigraphClique(D);
[ 5 ]
gap> DigraphMaximalClique(D);
[ 5 ]
gap> DigraphClique(D, [1, 2]);
[ 1, 2, 3 ]
gap> DigraphMaximalClique(D, [1, 3]);
[ 1, 3, 2 ]
gap> DigraphClique(D, [1], [2]);
```

---
DigraphMaximalClique(D, [1], [3, 4]);
fail

DigraphClique(D, [1, 5]);
fail

DigraphClique(D, [], [], 2);
[ 1, 2 ]

D := Digraph(IsMutableDigraph,
>     [[2, 3, 4], [1, 3], [1, 2], [1, 5], []]);
<mutable digraph with 5 vertices, 9 edges>

IsSymmetricDigraph(D);
false

DigraphClique(D);
[ 5 ]

8.1.4 DigraphMaximalClique

DigraphMaximalClique(digraph[, include[, exclude[, limit[, size]]]])) (function)
DigraphMaximalCliqueReps(digraph[, include[, exclude[, limit[, size]]]])) (function)

DigraphClique(digraph[, include[, exclude[, limit[, size]]]])) (function)
DigraphCliqueReps(digraph[, include[, exclude[, limit[, size]]]])) (function)
DigraphMaximalClique(digraph) (attribute)
DigraphMaximalCliqueReps(digraph) (attribute)

Returns:  An immutable list of lists of positive integers.

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

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

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

Let G denote the automorphism group of maximal symmetric subdigraph of digraph without loops (see AutomorphismGroup (7.2.2) and MaximalSymmetricSubdigraphWithoutLoops (3.3.5)).

Distinct cliques

DigraphMaximalClique and DigraphClique each return a duplicate-free list of at most limit cliques of digraph that satisfy the arguments.

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

Orbit representatives of cliques

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

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

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

Example

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

Example

```
gap> D := Digraph(IsMutableDigraph, [  
> [2, 3], [1, 3], [1, 2, 4], [3, 5, 6], [4, 6], [4, 5]];  
<mutable digraph with 6 vertices, 14 edges>  
gap> IsSymmetricDigraph(D);  
true  
gap> G := AutomorphismGroup(D);  
Group([ (5,6), (1,2), (1,5)(2,6)(3,4) ])  
gap> DigraphMaximalCliques(D);  
[ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 3, 4 ] ]  
```

8.1.5 CliqueNumber

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

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

If \( \text{digraph} \) is a digraph, then \( \text{CliqueNumber(digraph)} \) 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

```
gap> D := CompleteDigraph(4);  
gap> CliqueNumber(D);  
```

gap> D := Digraph([[1, 2, 4, 4], [1, 3, 4], [2, 1], [1, 2]]);
<immutable multidigraph with 4 vertices, 11 edges>
gap> CliqueNumber(D);
3
gap> D := CompleteDigraph(IsMutableDigraph, 4);

8.2 Finding independent sets

8.2.1 IsIndependentSet

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

Returns: true or false.

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

\[ \text{gap> D := CycleDigraph(4)}; \]
\[ \text{gap> IsIndependentSet(D, [1])}; \]
true
\[ \text{gap> IsMaximalIndependentSet(D, [1])}; \]
false
\[ \text{gap> IsIndependentSet(D, [1, 4, 3])}; \]
false
\[ \text{gap> IsIndependentSet(D, [2, 4])}; \]
true
\[ \text{gap> IsMaximalIndependentSet(D, [2, 4])}; \]
true
\[ \text{gap> D := CycleDigraph(IsMutableDigraph, 4)}; \]
\[ \text{gap> IsIndependentSet(D, [1])}; \]
true

8.2.2 DigraphIndependentSet

\[ \text{DigraphIndependentSet}(\text{digraph}[, \text{include}[, \text{exclude}[, \text{size}]]]) \] (function)
\[ \text{DigraphMaximalIndependentSet}(\text{digraph}[, \text{include}[, \text{exclude}[, \text{size}]]]) \] (function)

Returns: An immutable list of positive integers, or fail.

If \text{digraph} is a digraph, then these functions returns an independent set of \text{digraph} if one exists that satisfies the arguments, else it returns fail. An independent set is defined by the set of vertices that it contains; see IsIndependentSet (8.2.1) and IsMaximalIndependentSet (8.2.1).

The optional arguments \text{include} and \text{exclude} must each be a (possibly empty) duplicate-free list of vertices of \text{digraph}, and the optional argument \text{size} must be a positive integer. By default,
include and exclude are empty. These functions will search for an independent set of digraph that includes the vertices of include but 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.

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

---

**Example**

```gap
gap> D := ChainDigraph(6);
<immutable chain digraph with 6 vertices>
gap> DigraphIndependentSet(D);
[ 6, 4, 2 ]
gap> DigraphMaximalIndependentSet(D);
[ 6, 4, 2 ]
gap> DigraphIndependentSet(D, [2, 4]);
[ 2, 4, 6 ]
gap> DigraphMaximalIndependentSet(D, [1, 3]);
[ 1, 3, 6 ]
gap> DigraphIndependentSet(D, [2, 4], [6]);
[ 2, 4 ]
gap> DigraphMaximalIndependentSet(D, [2, 4], [6]); fail
gap> DigraphIndependentSet(D, [1], [], 2);
[ 1, 3 ]
gap> DigraphMaximalIndependentSet(D, [1], [], 2); fail
gap> DigraphMaximalIndependentSet(D, [1], [], 3);
[ 1, 3, 5 ]
gap> D := ChainDigraph(IsMutableDigraph, 6);
<mutable digraph with 6 vertices, 5 edges>
```
Digraphs

\begin{verbatim}
gap> DigraphIndependentSet(D);
[ 6, 4, 2 ]
gap> DigraphMaximalIndependentSet(D);
[ 6, 4, 2 ]
gap> DigraphIndependentSet(D, [2, 4]);
[ 2, 4, 6 ]
gap> DigraphMaximalIndependentSet(D, [1, 3]);
[ 1, 3, 6 ]
gap> DigraphIndependentSet(D, [2, 4], [6]);
fail
gap> DigraphMaximalIndependentSet(D, [2, 4], [6]);
fail
gap> DigraphIndependentSet(D, [1], [], 2);
[ 1, 3 ]
gap> DigraphMaximalIndependentSet(D, [1], [], 2);
fail
gap> DigraphMaximalIndependentSet(D, [1], [], 3);
[ 1, 3, 5 ]
\end{verbatim}

8.2.3 DigraphMaximalIndependentSets

- DigraphMaximalIndependentSets(digraph[, include[, exclude[, limit[, size]]]])
- DigraphMaximalIndependentSetsReps(digraph[, include[, exclude[, limit[, size]]]])
- DigraphIndependentSets(digraph[, include[, exclude[, limit[, size]]]])
- DigraphIndependentSetsReps(digraph[, include[, exclude[, limit[, size]]]])
- DigraphMaximalIndependentSetsAttr(digraph)
- DigraphMaximalIndependentSetsRepsAttr(digraph)

**Returns:** An immutable 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 that 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 chosen to be empty lists, and `limit` is set to infinity.

The functions will return as many suitable independent sets as possible, up to the number `limit`. These functions will find independent sets that contain all of the vertices of `include` but do not contain any of the vertices of `exclude` The argument `size` restricts the search to those cliques that 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 (3.3.11) 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-
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 limit orbits representatives (under the action of $G$ on sets of vertices) of independent sets of digraph that satisfy the arguments.

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

Example

```
gap> D := CycleDigraph(5);
<immutable cycle digraph with 5 vertices>
gap> DigraphMaximalIndependentSetsReps(D);
[ [ 1, 3 ] ]
gap> DigraphIndependentSetsReps(D);
[ [ 1 ], [ 1, 3 ] ]
gap> Set(DigraphMaximalIndependentSets(D));
[ [ 1, 3 ], [ 1, 4 ], [ 2, 4 ], [ 2, 5 ], [ 3, 5 ] ]
gap> DigraphMaximalIndependentSets(D, [1]);
[ [ 1, 3 ] ]
gap> DigraphIndependentSets(D, [], [4, 5]);
[ [ 1 ], [ 2 ], [ 3 ], [ 1, 3 ] ]
gap> DigraphIndependentSets(D, [], [4, 5], 1, 2);
[ [ 1, 3 ] ]
gap> D := CycleDigraph(IsMutableDigraph, 5);
<mutable digraph with 5 vertices, 5 edges>
gap> DigraphMaximalIndependentSetsReps(D);
[ [ 1, 3 ] ]
gap> DigraphIndependentSetsReps(D);
[ [ 1 ], [ 1, 3 ] ]
gap> Set(DigraphMaximalIndependentSets(D));
[ [ 1, 3 ], [ 1, 4 ], [ 2, 4 ], [ 2, 5 ], [ 3, 5 ] ]
gap> DigraphMaximalIndependentSets(D, [1]);
[ [ 1, 3 ] ]
gap> DigraphIndependentSets(D, [], [4, 5]);
[ [ 1 ], [ 2 ], [ 3 ], [ 1, 3 ] ]
gap> DigraphIndependentSets(D, [], [4, 5], 1, 2);
[ [ 1, 3 ] ]
```
Chapter 9

Visualising and IO

9.1 Visualising a digraph

9.1.1 Splash

\[\text{Splash}(str[, opts])\]

\textbf{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 \texttt{dot} or \LaTeX{} text file and you must have \texttt{GraphViz} and \texttt{pdflatex} installed on your computer. For details about these file formats, see \url{http://www.latex-project.org} and \url{http://www.graphviz.org}.

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

The optional second argument \(opts\) should be a record with components corresponding to various options, given below.

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

\textbf{directory} this should be a string representing the name of the directory in \texttt{path} where you want \texttt{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 \texttt{path} is present, and the result of \texttt{DirectoryTemporary} (Reference: \texttt{DirectoryTemporary}) is used otherwise.

\textbf{filename} this should be a string representing the name of the file where \texttt{str} will be written. The default value of this option is "vizpicture".

\textbf{viewer} this should be a string representing the name of the program which should open the files produced by \texttt{GraphViz} or \texttt{pdflatex}.

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

\begin{verbatim}
gap> Splash(DotDigraph(RandomDigraph(4)));
\end{verbatim}

9.1.2 DotDigraph
\begin{verbatim}
▷ DotDigraph(digraph) (attribute)
▷ DotVertexLabelledDigraph(digraph) (operation)

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 ver-
tices, 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.

The string returned by DotDigraph or DotVertexLabelledDigraph can be written to a file using
the command FileString (GAPDoc: FileString).
\end{verbatim}

Example

\begin{verbatim}
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);
<immutable digraph with 4 vertices, 15 edges>
gap> FileString("dot/k4.dot", DotDigraph(gr));
154
\end{verbatim}

9.1.3 DotSymmetricDigraph
\begin{verbatim}
▷ DotSymmetricDigraph(digraph) (attribute)

Returns: A string.

This function produces a graphical representation of the symmetric digraph digraph.
DotSymmetricDigraph will return an error if digraph is not a symmetric digraph. See
IsSymmetricDigraph (6.1.12).
\end{verbatim}
Vertices are displayed as circles, numbered consistently with digraph. Since digraph is symmetric, for every non-loop edge there is a complementary edge with opposite source and range. DotSymmetricDigraph displays each pair of complementary edges as a single line between the relevant vertices, with no arrowhead.

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

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

Example

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

**9.1.4 DotPartialOrderDigraph**

▷ DotPartialOrderDigraph(digraph)  
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 (6.1.16).

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 (3.3.13) 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 (GAPDoc: FileString).

Example

```gap
gap> poset := Digraph([[1, 4], [2], [2, 3, 4], [4]]);
gap> IsPartialOrderDigraph(gr);
true
gap> FileString("dot/poset.dot", DotPartialOrderDigraph(gr));
83
```

**9.1.5 DotPreorderDigraph**

▷ DotPreorderDigraph(digraph)  
▷ DotQuasiorderDigraph(digraph)  
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 (6.1.15).

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 (3.3.8) Q
obtained by forming the quotient of digraph by the partition of its vertices into the strongly connected components satisfies \textit{IsPartialOrderDigraph} (6.1.16). Thus every vertex of $Q$ corresponds to a strongly connected component of digraph. The output of \texttt{DotPreorderDigraph} displays the \textit{DigraphReflexiveTransitiveReduction} (3.3.13) 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 \url{http://www.graphviz.org}.

The string returned by \texttt{DotPreorderDigraph} can be written to a file using the command \texttt{FileString} (\texttt{GAPDoc: FileString}).

\begin{verbatim}
gap> preset := Digraph([[1, 2, 4, 5], [1, 2, 4, 5], [3, 4], [4], [1, 2, 4, 5]]);
gap> IsPreorderDigraph(gr); true
gap> FileString("dot/preset.dot", DotPreorderDigraph(gr));
83
\end{verbatim}

### 9.1.6 DotHighlightedDigraph

\begin{verbatim}
> DotHighlightedDigraph(digraph, verts[, colour1, colour2])
\end{verbatim}

\textbf{Returns:} A string.

\texttt{DotHighlightedDigraph} produces a graphical representation of the digraph \texttt{digraph}, where the vertices in the list \texttt{verts}, and edges between them, are drawn with colour \texttt{colour1} and all other vertices and edges in \texttt{digraph} are drawn with colour \texttt{colour2}. If \texttt{colour1} and \texttt{colour2} are not given then \texttt{DotHighlightedDigraph} uses black and grey respectively.

Note that \texttt{DotHighlightedDigraph} does not validate the colours \texttt{colour1} and \texttt{colour2} - consult the GraphViz documentation to see what is available. See \texttt{DotDigraph} (9.1.2) for more details on the output.

\begin{verbatim}
gap> digraph := Digraph([[2, 3], [2], [1, 3]]);
<digraph with 3 vertices, 5 edges>
gap> FileString("dot/my_digraph.dot",
> DotHighlightedDigraph(digraph, [1, 2], "red", "black"));
264
\end{verbatim}

### 9.2 Reading and writing digraphs to a file

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

\textbf{Graph6}

\textit{Graph6} is a graph data format for storing undirected graphs with no multiple edges nor loops of size up to $2^{16} - 1$ 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 \textit{Graph6}.

\textbf{Sparse6}

\textit{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 returns a more compact encoding than Graph6 for sparse graphs, i.e., graphs where the number of edges is much less than the number of vertices squared. For a more detailed 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([filt, ]str) (operation)
- DigraphFromDigraph6String([filt, ]str) (operation)
- DigraphFromSparse6String([filt, ]str) (operation)
- DigraphFromDiSparse6String([filt, ]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.

Each of these functions takes an optional first argument filt, which should be either IsMutableDigraph (3.1.2) or IsImmutableDigraph (3.1.3), and which specifies whether the output digraph shall be mutable or immutable. If no first argument is provided, then an immutable digraph is returned by default.

---

**Example**

```gap
gap> DigraphFromGraph6String("?");
<immutable empty digraph with 0 vertices>
gap> DigraphFromGraph6String("C");
<immutable digraph with 4 vertices, 8 edges>
gap> DigraphFromGraph6String("H?AAEM{";
<immutable digraph with 9 vertices, 22 edges>
gap> DigraphFromDigraph6String("&?");
<immutable empty digraph with 0 vertices>
gap> DigraphFromDigraph6String(IsMutableDigraph, "&D000W?";)
<mutable digraph with 5 vertices, 5 edges>
gap> DigraphFromDiSparse6String(".CaWBGA?b");
<immutable multidigraph with 4 vertices, 9 edges>
```
9.2.2 Graph6String

- Graph6String(digraph) (operation)
- Digraph6String(digraph) (operation)
- Sparse6String(digraph) (operation)
- DiSparse6String(digraph) (operation)

Returns: A string.
These four functions return a highly compressed string fully describing the digraph digraph.
Graph6 and Digraph6 are formats best used on small, dense graphs, if applicable. For larger, sparse graphs use Sparse6 and DiSparse6 (this latter also preserves multiple edges).
See WriteDigraphs (9.2.5).

Example

```gap
gap> gr := Digraph([[2, 3], [1], [1]]);
<immutable 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 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 coder is not specified, then the encoding of the digraphs in the returned file object must be specified in the the file extension. The file extension must be one of:.g6, .s6, .d6, .ds6, .txt, .p, or .pickle; more details of these file formats is given below.
If the optional argument mode is specified, then it must be one of: "w" (for write), "a" (for append), or "r" (for read). If mode is not specified, then "r" is used by default.
If filename ends in one of:.gz, .bz2, or .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 DigraphFile can be given as the first argument for either of the functions ReadDigraphs (9.2.4) or WriteDigraphs (9.2.5). The purpose of this is to reduce the overhead of recreating the file object inside the functions ReadDigraphs (9.2.4) or 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:

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 (7.2.10) is non-trivial.

Example

```gap
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);

gap> file := DigraphFile(filename, "r");
gap> ReadDigraphs(file, 9);
<immutable 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 nth 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 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 (7.2.10) is non-trivial.

---

**Example**

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

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 (7.2.10) is non-trivial.

```gap
gap> grs := [];;
gap> grs[1] := Digraph([]);
<immutable empty digraph with 0 vertices>
gap> grs[2] := Digraph([[1, 3], [2], [1, 2]]);
<immutable 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]]);
<immutable 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);
[ <immutable empty digraph with 0 vertices>,
  <immutable digraph with 3 vertices, 5 edges>,
  <immutable digraph with 10 vertices, 51 edges> ]
```

### 9.2.6 IteratorFromDigraphFile

**IteratorFromDigraphFile**

- **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)`.  

```gap
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);;
gap> iter := IteratorFromDigraphFile(filename);
<iterator>
gap> for x in iter do od;
```
9.2.7 DigraphPlainTextLineEncoder

- DigraphPlainTextLineEncoder(delimiter1[, delimiter2], offset) (function)
- DigraphPlainTextLineDecoder(delimiter1[, delimiter2], offset) (operation)

**Returns:** A string.

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

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

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

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

Example

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

9.2.8 TournamentLineDecoder

- TournamentLineDecoder(str) (operation)

**Returns:** A digraph.

This function takes a string *str*, decodes it, and then returns the tournament [see *IsTournament* (6.1.13)] 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
gap> gr := TournamentLineDecoder("100001");
<immutable digraph with 4 vertices, 6 edges>
gap> OutNeighbours(gr);
[ [ 2 ], [ ], [ 1, 2, 4 ], [ 1, 2 ] ]
```
9.2.9  AdjacencyMatrixUpperTriangleLineDecoder

▷ AdjacencyMatrixUpperTriangleLineDecoder(str)  (operation)

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
gap> gr := AdjacencyMatrixUpperTriangleLineDecoder("100001");
<immutable digraph with 4 vertices, 2 edges>
gap> OutNeighbours(gr);
[ [ 2 ], [ ], [ 4 ], [ ] ]
gap> gr := AdjacencyMatrixUpperTriangleLineDecoder("111111x111");
<immutable digraph with 5 vertices, 9 edges>
gap> OutNeighbours(gr);
[ [ 2, 3, 4, 5 ], [ 3, 4 ], [ 4, 5 ], [ 5 ], [ ] ]
```

9.2.10  TCodeDecoder

▷ TCodeDecoder(str)  (operation)

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

The operation DigraphFromPlainTextString takes an optional first argument IsMutableDigraph (3.1.2) or IsImmutableDigraph (3.1.3), which specifies whether the output digraph shall be mutable or immutable. If no first argument is provided, then an immutable digraph is returned by default.

Example

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

9.2.12 WritePlainTextDigraph

« WritePlainTextDigraph(filename, digraph, delimiter, offset) » (function)
« ReadPlainTextDigraph(filename, delimiter, offset, ignore) » (operation)

These functions write and read a single digraph in a human-readable plain text format as follows: each line contains a single edge, and each edge is written as a pair of integers separated by the string delimiter.

filename should be the name of a file which will be written to or read from, and offset should be an integer which is added to each vertex number as it is written or read. For example, if WritePlainTextDigraph is called with offset -1, then the vertices will be numbered in the file starting from 0 instead of 1 - ReadPlainTextDigraph would then need to be called with offset 1 to convert back to the original graph.

ignore should be a list of characters which will be ignored when reading the graph.

Example

```gap
gap> gr := Digraph([[1, 2, 3], [1, 1], [2]]);
<immutable multidigraph with 3 vertices, 6 edges>
gap> filename := Concatenation(DIGRAPHS_Dir(), "/tst/out/plain.txt");;
gap> WritePlainTextDigraph(filename, gr, ",", -1);
gap> ReadPlainTextDigraph(filename, ",", 1, ["/", "]"]);
<immutable multidigraph with 3 vertices, 6 edges>
```
9.2.13 WriteDIMACSDigraph

\[\text{WriteDIMACSDigraph}(\text{filename}, \text{digraph})\] (operation)

\[\text{ReadDIMACSDigraph}(\text{filename})\] (operation)

These operations write or read the single symmetric digraph \textit{digraph} to or from a file in DIMACS format, as appropriate. The operation \text{WriteDIMACSDigraph} records the vertices and edges of \textit{digraph}. The vertex labels of \textit{digraph} will be recorded only if they are integers. See \text{IsSymmetricDigraph} (6.1.12) and \text{DigraphVertexLabels} (5.1.9).

The first argument \textit{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 \textit{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 \textit{c} and followed by any number of characters is a comment line, and is ignored.

- A line beginning with \textit{p} defines the numbers of vertices and edges the digraph. This line has the format \textit{p edge <nr_vertices> <nr_edges>}, where \textit{<nr_vertices>} and \textit{<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. Although it is required to be present, the value of \textit{<nr_edges>} will be ignored. The correct number of edges will be deduced from the rest of the information in the file.

- A line of the form \textit{e <v> <w>}, where \textit{<v>} and \textit{<w>} are integers in the range \([1 .. <nr_vertices>]\), specifies that there is a (symmetric) edge in the digraph between the vertices \textit{<v>} and \textit{<w>}. A symmetric edge only needs to be defined once; an additional line \textit{e <v> <w>}, or \textit{e <w> <v>}, will be interpreted as an additional, multiple, edge. Loops are permitted.

- A line of the form \textit{n <v> <label>}, where \textit{<v>} is an integer in the range \([1 .. <nr_vertices>]\) and \textit{<label>} is an integer, signifies that the vertex \textit{<v>} has the label \textit{<label>} in the digraph. If a label is not specified for a vertex, then \text{ReadDIMACSDigraph} will assign the label 1, according to the DIMACS specification.

A detailed definition of the DIMACS format can be found at \url{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

\begin{verbatim}
gap> gr := Digraph([[2], [1, 3, 4], [2, 4], [2, 3]]);
<immutable digraph with 4 vertices, 8 edges>
gap> filename := Concatenation(DIGRAPHS_Dir(), > "~/tst/out/dimacs.dimacs");
gap> WriteDIMACSDigraph(filename, gr);
gap> ReadDIMACSDigraph(filename);
<immutable digraph with 4 vertices, 8 edges>
\end{verbatim}
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 (3.1.7)</td>
</tr>
<tr>
<td>EdgeOrbitsGraph</td>
<td>EdgeOrbitsDigraph (3.1.10)</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 (3.1.12)</td>
</tr>
<tr>
<td>AddEdgeOrbit</td>
<td>DigraphAddEdgeOrbit (3.3.17)</td>
</tr>
<tr>
<td>RemoveEdgeOrbit</td>
<td>DigraphRemoveEdgeOrbit (3.3.22)</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 (5.3.10)</td>
</tr>
<tr>
<td>ConnectedComponents</td>
<td>DigraphConnectedComponents (5.3.9)</td>
</tr>
<tr>
<td>Bicomponents</td>
<td>DigraphBicomponents (5.3.13)</td>
</tr>
<tr>
<td>DistanceSet</td>
<td>DigraphDistanceSet (5.3.5)</td>
</tr>
<tr>
<td>Layers</td>
<td>DigraphLayers (5.3.23)</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 (3.3.3)</td>
</tr>
<tr>
<td>DistanceSetInduced</td>
<td>None</td>
</tr>
<tr>
<td>DistanceGraph</td>
<td>DistanceDigraph (3.3.39)</td>
</tr>
<tr>
<td>ComplementGraph</td>
<td>DigraphDual (3.3.10)</td>
</tr>
<tr>
<td>PointGraph</td>
<td>None</td>
</tr>
<tr>
<td>EdgeGraph</td>
<td>EdgeUndirectedDigraph (3.3.35)</td>
</tr>
<tr>
<td>SwitchedGraph</td>
<td>None</td>
</tr>
<tr>
<td>UnderlyingGraph</td>
<td>DigraphSymmetricClosure (3.3.11)</td>
</tr>
<tr>
<td>QuotientGraph</td>
<td>QuotientDigraph (3.3.8)</td>
</tr>
<tr>
<td>BipartiteDouble</td>
<td>BipartiteDoubleDigraph (3.3.37)</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 (7.3.15)</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 (7.2.17)</td>
</tr>
<tr>
<td>IsIsomorphicGraph</td>
<td>IsIsomorphicDigraph (7.2.15)</td>
</tr>
<tr>
<td>GraphIsomorphismClassRepresentatives</td>
<td>None</td>
</tr>
</tbody>
</table>
References


Index

< (for digraphs), 51
= (for digraphs), 51
AdjacencyMatrix, 62
AdjacencyMatrixMutableCopy, 62
AdjacencyMatrixUpperTriangleLine-Decoder, 167
ArticulationPoints, 79
AsBinaryRelation, 17
AsDigraph, 17
AsGraph, 19
AsMonoid, 88
AsSemigroup
  for a filter and a digraph, 88
  for a filter, semilattice digraph, and two lists, 89
AsTransformation, 19
AutomorphismGroup
  for a digraph, 118
  for a digraph and a homogeneous list, 119
  for a digraph, homogeneous list, and list, 120
BipartiteDoubleDigraph, 43
BlissAutomorphismGroup
  for a digraph, 118
  for a digraph and homogenous list, 118
  for a digraph, homogenous list, and list, 118
BlissCanonicalDigraph, 123
BlissCanonicalLabelling
  for a digraph, 121
  for a digraph and a list, 122
BooleanAdjacencyMatrix, 63
BooleanAdjacencyMatrixMutableCopy, 63
CayleyDigraph, 16
ChainDigraph, 47
CharacteristicPolynomial, 63
ChromaticNumber, 143
CliqueNumber, 151
CliquesFinder, 147
CompleteBipartiteDigraph, 47
CompleteDigraph, 47
CompleteMultipartiteDigraph, 48
CycleDigraph, 48
DegreeMatrix, 70
Digraph, 12
  for a group, list, function, and function, 12
  for a list and function, 12
Digraph6String, 161
DigraphAddAllLoops, 43
DigraphAddAllLoopsAttr, 43
DigraphAddEdge
  for a digraph and an edge, 30
  for a digraph, source, and range, 30
DigraphAddEdgeOrbit, 31
DigraphAddEdges, 32
DigraphAddVertex, 29
DigraphAddVertices
  for a digraph and a list of labels, 30
  for a digraph and an integer, 30
DigraphAdjacencyFunction, 64
DigraphAllSimpleCircuits, 83
DigraphBicomponents, 78
DigraphByAdjacencyMatrix, 14
DigraphByEdges, 15
DigraphByInNeighbors, 16
DigraphByInNeighbours, 16
DigraphByOutNeighboursType, 12
DigraphCartesianProduct
  for a list of digraphs, 39
  for a positive number of digraphs, 39
DigraphCartesianProductProjections, 40
DigraphClique, 149
DigraphCliques, 150
DigraphCliquesReps, 150
DigraphClosure, 44
DigraphColouring
  for a digraph and a number of colours, 142
DigraphConnectedComponent, 77
DigraphConnectedComponents, 76
DigraphCopy, 20
DigraphCopySameMutability, 20
DigraphCore, 143
DigraphDegeneracy, 85
DigraphDegeneracyOrdering, 85
DigraphDiameter, 71
DigraphDirectProduct
   for a list of digraphs, 40
   for a positive number of digraphs, 40
DigraphDirectProductProjections, 41
DigraphDisjointUnion
   for a list of digraphs, 37
   for an arbitrary number of digraphs, 37
DigraphDistanceSet
   for a digraph, a pos int, and a list, 74
   for a digraph, a pos int, and an int, 74
DigraphDual, 26
DigraphDualAttr, 26
DigraphEdgeLabel, 59
DigraphEdgeLabels, 59
DigraphEdges, 55
DigraphEdgeUnion
   for a list of digraphs, 38
   for a positive number of digraphs, 38
DigraphEmbedding, 137
DigraphEpimorphism, 137
DigraphFamily, 12
DigraphFile, 161
DigraphFloydWarshall, 80
DigraphFromDigraph6String, 160
DigraphFromDiSparse6String, 160
DigraphFromGraph6String, 160
DigraphFromPlainTextString, 168
DigraphFromSparse6String, 160
DigraphGirth, 74
DigraphGreedyColouring
   for a digraph, 142
   for a digraph and vertex order, 142
   for a digraph and vertex order function, 142
DigraphGroup, 124
DigraphHasLoops, 95
DigraphHomomorphism, 135
DigraphImmutableCopy, 20
DigraphImmutableCopyIfImmutable, 20
DigraphImmutableCopyIfMutable, 20
DigraphIndependentSet, 152
DigraphIndependentSets, 154
DigraphIndependentSetsReps, 154
DigraphInEdges, 60
DigraphJoin
   for a list of digraphs, 38
   for a positive number of digraphs, 38
DigraphLayers, 84
DigraphLongestDistanceFromVertex, 73
DigraphLongestSimpleCircuit, 84
DigraphLoops, 69
DigraphMaximalClique, 149
DigraphMaximalCliqueAttr, 150
DigraphMaximalCliqueReps, 150
DigraphMaximalCliqueRepsAttr, 150
DigraphMaximalIndependentSet, 152
DigraphMaximalIndependentSets, 154
DigraphMaximalIndependentSetsAttr, 154
DigraphMaximalIndependentSetsReps, 154
DigraphMaximalIndependentSetsRepsAttr, 154
DigraphMonomorphism, 136
DigraphMutableCopy, 20
DigraphMutableCopyIfImmutable, 20
DigraphMutableCopyIfMutable, 20
DigraphMycielskian, 44
DigraphMycielskianAttr, 44
DigraphNrConnectedComponents, 76
DigraphNrEdges, 55
DigraphNrStronglyConnectedComponents, 77
DigraphNrVertices, 54
DigraphOddGirth, 75
DigraphOrbitReps, 125
DigraphOrbits, 125
DigraphOutEdges, 60
DigraphPath, 81
DigraphPeriod, 80
DigraphPlainTextLineDecoder, 166
DigraphPlainTextLineEncoder, 166
DigraphRange, 65
DigraphReflexiveTransitiveClosure, 27
DigraphReflexiveTransitiveClosureAttr, 27
DigraphReflexiveTransitiveReduction, 28
DigraphReflexiveTransitiveReductionAttr, 28
DigraphRemoveAllMultipleEdges, 36
DigraphRemoveAllMultipleEdgesAttr, 36
DigraphRemoveEdge
  for a digraph and an edge, 34
  for a digraph, source, and range, 34
DigraphRemoveEdgeOrbit, 34
DigraphRemoveEdges, 35
DigraphRemoveLoops, 35
DigraphRemoveLoopsAttr, 35
DigraphRemoveVertices, 32
DigraphReverse, 26
DigraphReverseAttr, 26
DigraphReverseEdge
  for a digraph and an edge, 36
  for a digraph, source, and range, 36
DigraphReverseEdges
  for a digraph and a list of edges, 36
DigraphReflexiveTransitiveReductionAttr, 28
DigraphRemoveAllMultipleEdgesAttr, 36
DigraphRemoveEdge
  for a digraph and an edge, 34
  for a digraph, source, and range, 34
DigraphRemoveEdgeOrbit, 34
DigraphRemoveEdges, 35
DigraphRemoveLoops, 35
DigraphRemoveLoopsAttr, 35
DigraphRemoveVertices, 32
DigraphReverse, 26
DigraphReverseAttr, 26
DigraphReverseEdge
  for a digraph and an edge, 36
  for a digraph, source, and range, 36
DigraphReverseEdges
  for a digraph and a list of edges, 36
Digraphs package overview, 5
DigraphSchreierVector, 126
DigraphShortestDistance
  for a digraph and a list, 72
  for a digraph and two vertices, 72
  for a digraph, a list, and a list, 72
DigraphShortestDistances, 73
DigraphShortestPath, 82
DigraphSinks, 56
DigraphsMakeDoc, 9
DigraphSource, 65
DigraphSources, 56
DigraphsRespectsColouring, 140
DigraphStabilizer, 127
DigraphTestExtreme, 9
DigraphTestInstall, 9
DigraphTestStandard, 9
DigraphStronglyConnectedComponent, 78
DigraphStronglyConnectedComponents, 77
DigraphsUseBliss, 117
DigraphsUseNauty, 117
DigraphSymmetricClosure, 27
DigraphSymmetricClosureAttr, 27
DigraphTopologicalSort, 56
DigraphTransitiveClosure, 27
DigraphTransitiveClosureAttr, 27
DigraphTransitiveReduction, 28
DigraphTransitiveReductionAttr, 28
DigraphUndirectedGirth, 75
DigraphVertexLabel, 57
DigraphVertexLabels, 58
DigraphVertices, 54
DigraphWelshPowellOrder, 143
DiSparse6String, 161
DistanceDigraph
  for digraph and int, 43
  for digraph and list, 43
DotDigraph, 157
DotHighlightedDigraph, 159
DotPartialOrderDigraph, 158
DotPreorderDigraph, 158
DotQuasiorderDigraph, 158
DotSymmetricDigraph, 157
DotVertexLabelledDigraph, 157
DoubleDigraph, 42
EdgeDigraph, 42
EdgeOrbitsDigraph, 15
EdgeUndirectedDigraph, 42
EmbeddingsDigraphs, 138
EmbeddingsDigraphsRepresentatives, 138
EmptyDigraph, 49
EpimorphismsDigraphs, 137
EpimorphismsDigraphsRepresentatives, 137
GeneralisedPetersenGraph, 50
GeneratorsOfCayleyDigraph, 87
GeneratorsOfEndomorphismMonoid, 141
GeneratorsOfEndomorphismMonoidAttr, 141
Graph, 18
Graph6String, 161
GroupOfCayleyDigraph, 87
HamiltonianPath, 86
HomomorphismDigraphsFinder, 133
HomomorphismsDigraphs, 135
HomomorphismsDigraphsRepresentatives, 135
InDegreeOfVertex, 68
InDegrees, 67
InDegreeSequence, 67
Digraphs

InDegreeSet, 67
InducedSubdigraph, 21
InNeighbors, 66
InNeighborsMutableCopy, 66
InNeighborsOfVertex, 69
InNeighbours, 66
InNeighboursMutableCopy, 66
InNeighborsOfVertex, 69
IsAcyclicDigraph, 106
IsAntiSymmetricDigraph, 95
IsAntisymmetricDigraph, 95
IsAperiodicDigraph, 109
IsBiconnectedDigraph, 107
IsBipartiteDigraph, 96
IsCayleyDigraph, 12
IsChainDigraph, 106
IsClique, 146
IsCompleteBipartiteDigraph, 96
IsCompleteDigraph, 97
IsCompleteMultipartiteDigraph, 97
IsConnectedDigraph, 107
IsCycleDigraph, 112
IsDigraph, 11
IsDigraphAutomorphism, 131
for a digraph and a transformation or permutation, 131
IsDigraphColouring, 132
for a transformation, 132
IsDigraphCore, 114
IsDigraphEmbedding, 140
for digraphs and a permutation or transformation, 140
IsDigraphEndomorphism, 139
for digraphs and a permutation or transformation, 138
IsDigraphEpimorphism, 138
for digraphs and a permutation or transformation, 138
IsDigraphHomomorphism, 138
for digraphs and a permutation or transformation, 138
IsDigraphIsomorphism, 131
for digraphs and transformation or permutation, 131
IsDigraphMonomorphism, 138
for digraphs and a permutation or transformation, 138
IsDigraphWithAdjacencyFunction, 12
IsDirectedTree, 109
IsDistanceRegularDigraph, 105
IsEdgeTransitive, 114
IsEmptyDigraph, 98
IsEquivalenceDigraph, 98
IsEulerianDigraph, 110
IsFunctionalDigraph, 99
IsHamiltonianDigraph, 111
IsImmutableDigraph, 11
IsIndependentSet, 152
IsInRegularDigraph, 104
IsIsomorphicDigraph
for digraphs, 127
for digraphs and homogeneous lists, 128
IsJoinSemilatticeDigraph, 103
IsLatticeDigraph, 103
IsMatching, 61
IsMaximalClique, 146
IsMaximalIndependentSet, 152
IsMaximalMatching, 61
IsMeetSemilatticeDigraph, 103
IsMultiDigraph, 99
IsMutableDigraph, 11
IsNullDigraph, 98
IsomorphismDigraphs
for digraphs, 128
for digraphs and homogeneous lists, 129
IsOuterPlanarDigraph, 113
IsOuterRegularDigraph, 104
IsPartialOrderDigraph, 103
IsPerfectMatching, 61
IsPlanarDigraph, 112
IsPreorderDigraph, 102
IsQuasiorderDigraph, 102
IsReachable, 81
IsReflexiveDigraph, 100
IsRegularDigraph, 105
IsStronglyConnectedDigraph, 108
IsSubdigraph, 51
IsSymmetricDigraph, 108
IsTournament, 101
Digraphs

IsTransitiveDigraph, 101
IsUndirectedForest, 110
IsUndirectedSpanningForest, 52
IsUndirectedSpanningTree, 52
IsUndirectedTree, 110
IsVertexTransitive, 114
IteratorFromDigraphFile, 165
IteratorOfPaths, 83
JohnsonDigraph, 49
KuratowskiOuterPlanarSubdigraph, 90
KuratowskiPlanarSubdigraph, 89
LaplacianMatrix, 71
LineDigraph, 42
LineUndirectedDigraph, 42
MaximalAntiSymmetricSubdigraph, 23
MaximalAntiSymmetricSubdigraphAttr, 23
MaximalSymmetricSubdigraph, 22
MaximalSymmetricSubdigraphAttr, 22
MaximalSymmetricSubdigraphWithoutLoops, 22
MaximalSymmetricSubdigraphWithoutLoopsAttr, 22
MonomorphismsDigraphs, 136
MonomorphismsDigraphsRepresentatives, 136
NautyAutomorphismGroup, 119
NautyCanonicalDigraph, 123
NautyCanonicalLabelling
  for a digraph, 121
  for a digraph and a list, 122
NrSpanningTrees, 86
NullDigraph, 49
OnDigraphs
  for a digraph and a perm, 116
  for a digraph and a transformation, 116
OnMultiDigraphs, 117
  for a digraph, perm, and perm, 117
OutDegreeOfVertex, 68
OutDegrees, 67
OutDegreeSequence, 67
OutDegreeSet, 67
OuterPlanarEmbedding, 92
OutNeighbors, 65
OutNeighborsMutableCopy, 65
OutNeighborsOfVertex, 68
OutNeighbours, 65
OutNeighboursMutableCopy, 65
OutNeighboursOfVertex, 68
PartialOrderDigraphJoinOfVertices, 70
PartialOrderDigraphMeetOfVertices, 70
PetersenGraph, 50
PlainTextString, 168
PlanarEmbedding, 91
QuotientDigraph, 25
RandomDigraph, 45
RandomLattice, 46
RandomMultiDigraph, 45
RandomTournament, 46
ReadDigraphs, 162
ReadDIMACSDigraph, 169
ReadPlainTextDigraph, 168
ReducedDigraph, 22
ReducedDigraphAttr, 22
RepresentativeOutNeighbours, 130
SemigroupOfCayleyDigraph, 87
SetDigraphEdgeLabel, 59
SetDigraphEdgeLabels
  for a digraph and a function, 59
  for a digraph and a list of lists, 59
SetDigraphVertexLabel, 57
SetDigraphVertexLabels, 58
Sparse6String, 161
Splash, 156
SubdigraphHomeomorphicToK23, 93
SubdigraphHomeomorphicToK33, 93
SubdigraphHomeomorphicToK4, 93
TCodeDecoder, 167
TournamentLineDecoder, 166
UndirectedSpanningForest, 24
UndirectedSpanningForestAttr, 24
UndirectedSpanningTree, 24
UndirectedSpanningTreeAttr, 24
WriteDigraphs, 164
WriteDIMACSDigraph, 169
WritePlainTextDigraph, 168