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: https://jdbm.me

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.
7  Homomorphisms  120
   7.1 Acting on digraphs  .......................................................... 120
   7.2 Isomorphisms and canonical labellings  .................................. 121
   7.3 Homomorphisms of digraphs  .............................................. 137

8  Finding cliques and independent sets  149
   8.1 Finding cliques  .............................................................. 150
   8.2 Finding independent sets  .................................................. 156

9  Visualising and IO  160
   9.1 Visualising a digraph  ....................................................... 160
   9.2 Reading and writing digraphs to a file  .................................. 163

A  Grape to Digraphs Command Map  175
   A.1 Functions to construct and modify graphs  ............................ 175
   A.2 Functions to inspect graphs, vertices and edges  .................... 175
   A.3 Functions to determine regularity properties of graphs  ............. 176
   A.4 Some special vertex subsets of a graph  ............................... 176
   A.5 Functions to construct new graphs from old  .......................... 177
   A.6 Vertex-Colouring and Complete Subgraphs  ............................ 177
   A.7 Automorphism groups and isomorphism testing for graphs ............ 177

References  178

Index  179
Chapter 1

The Digraphs package

1.1 Introduction

This is the manual for version 1.3.1 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
• Maria Tsalakou,
• 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 $s, r : E^1 \rightarrow E^0$, 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 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 $(\emptyset)$. 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 4.10.0 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.

• ensure that the datastructures package version 0.2.5 or higher is available.

• 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.

• THIS STEP IS OPTIONAL: certain functions in Digraphs require the NautyTracesInterface package to be available. If you want to make use of these functions, please ensure that the NautyTracesInterface package version 0.2 or higher is available. If NautyTracesInterface is not available, then Digraphs can be used as normal with the exception that functions whose names contain “Nauty” will not work.

• download the package archive digraphs-1.3.1.tar.gz from the Digraph package webpage.

• unzip and untar the file, this should create a directory called digraphs-1.3.1.

• locate the pkg directory of your GAP directory, which contains the directories lib, doc and so on. Move the directory digraphs-1.3.1 into the pkg directory.

• it is necessary to compile the Digraphs package. Inside the pkg/digraphs-1.3.1 directory, type
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-1.3.1 directory, type

```bash
./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 architecture.
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. How-
ever, you might find it necessary, at some point, to rebuild the documentation. To rebuild the docu-
mentation, please use the function DigraphsMakeDoc (2.4.1).

2.4.1 DigraphsMakeDoc

DigraphsMakeDoc()

Returns: Nothing

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

2.5 Testing your installation

In this section we describe how to test that Digraphs is working as intended. To test that Dig-
graphs 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 Dig-
graphs 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()

Returns: true or false.

This function can be called without arguments to test your installation of Digraphs is working cor-
rectly. 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()

Returns: 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()

Returns: 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

```plaintext
> 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

```plaintext
> 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

```plaintext
> 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 \textbf{IsCayleyDigraph}

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

3.1.5 \textbf{IsDigraphWithAdjacencyFunction}

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

3.1.6 \textbf{DigraphByOutNeighboursType}

\texttt{DigraphByOutNeighboursType} (global variable) \quad \texttt{DigraphFamily} (family)

The type of all digraphs is \texttt{DigraphByOutNeighboursType}. The family of all digraphs is \texttt{DigraphFamily}.

3.1.7 \textbf{Digraph}

\texttt{Digraph([filt, ]obj[, source, range])} \quad \texttt{Digraph([filt, ]list, func)} \quad \texttt{Digraph([filt, ]G, list, act, adj)}

\textbf{Returns:} A digraph.

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.

\textbf{for a list (i.e. an adjacency list)}

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

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

\textbf{for three lists}

if \texttt{obj} is a duplicate-free list, and \texttt{source} and \texttt{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 \texttt{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 \( \text{Length(source)} \) edges. For each \( i \) in [1 .. \( \text{Length(source)} \)] 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 .. \( \text{Length(list)} \)] and an edge from vertex i to vertex j if and only if func(list[i], list[j]) returns true.

**for a group, a list, and two functions**

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

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

The action of the group G on the objects in list is stored in the attribute DigraphGroup (7.2.10), 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.

```gap
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>

gap> gr := Digraph("
a", "b", "c"; "a", "b");
<immutable digraph with 3 vertices, 1 edge>

gap> gr := Digraph(5, [1, 2, 2, 4, 1, 1], [2, 3, 5, 5, 1, 1]);
<immutable multidigraph with 5 vertices, 6 edges>

gap> Petersen := Graph(SymmetricGroup(5), [[1, 2]], OnSets,
> function(x, y) return Intersection(x, y) = []; end);

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

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

### 3.1.8 DigraphByAdjacencyMatrix

> DigraphByAdjacencyMatrix([filt, list])

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

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

```gap
gap> DigraphByAdjacencyMatrix([
> [0, 1, 0, 2, 0],
> [1, 1, 1, 0, 1],
> [0, 0, 0, 0, 2],
> [1, 1, 1, 0, 1],
> [0, 0, 0, 0, 2]]); 
<immutable digraph with 5 vertices, 10 edges>
```
Digraphs

3.1.9 DigraphByEdges

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

**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 \( m \) such that its list equal \( \text{list} \).

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

See \text{DigraphEdges} (5.1.3).

**Example**

\[
\begin{align*}
&\text{gap> } \text{DigraphByEdges}([1, 3], [2, 1], [2, 3], [2, 5], [3, 6], [4, 6], [5, 2], [5, 4], [5, 6], [6, 6])
\end{align*}
\]

<immutable digraph with 6 vertices, 10 edges>

\[
\begin{align*}
&\text{gap> } \text{DigraphByEdges}([1, 3], [2, 1], [2, 3], [2, 5], [3, 6], [4, 6], [5, 2], [5, 4], [5, 6], [6, 6], [6, 6]), 12)
\end{align*}
\]

<immutable digraph with 12 vertices, 10 edges>

\[
\begin{align*}
&\text{gap> } \text{DigraphByEdges}(\text{IsMutableDigraph},
\end{align*}
\]

<mutable digraph with 12 vertices, 10 edges>

3.1.10 EdgeOrbitsDigraph

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

**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 $[1 \ldots n]$ 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
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

- **DigraphByInNeighbours** ([filt, ]list) (operation)
- **DigraphByInNeighbors** ([filt, ]list) (operation)

**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 list is a list of lists of positive integers list the range $[1 \ldots \text{Length(list)}]$, then this function returns the digraph with vertices $E_0 = [1 \ldots \text{Length(list)}]$, and edges corresponding to the entries of list. More precisely, there is an edge with source vertex $i$ and range vertex $j$ if $i$ is listed $j$.

If $i$ occurs list $j$ with multiplicity $k$, then there are $k$ multiple edges from $i$ to $j$.

See InNeighbours (5.2.7).

Example

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

3.1.12 CayleyDigraph

- **CayleyDigraph** ($G$, gens) (operation)

**Returns:** An immutable digraph.

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

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
gap> G := DihedralGroup(8);
<pc group of size 8 with 3 generators>
gap> CayleyDigraph(G);
<immutable digraph with 8 vertices, 24 edges>
gap> G := DihedralGroup(IsPermGroup, 8);
Group([ (1,2,3,4), (2,4) ])
gap> CayleyDigraph(G);
<immutable digraph with 8 vertices, 16 edges>
gap> digraph := CayleyDigraph(G, [()]);
<immutable digraph with 8 vertices, 8 edges>
gap> GroupOfCayleyDigraph(digraph) = G;
true
gap> 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 $[1 \ldots n]$. The pair $[i,j]$ is in the binary relation if and only if $[i,j]$ is an edge in `digraph`.

```gap
gap> D := Digraph([[3, 2], [1, 2], [2], [3, 4]]);
<immutable digraph with 4 vertices, 7 edges>
gap> 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 \textit{trans} is a transformation, and \textit{n} is a non-negative integer such that the restriction of \textit{trans} to \([1 \ldots n]\) defines a transformation of \([1 \ldots n]\), then \textit{AsDigraph} returns the functional digraph with \textit{n} vertices defined by \textit{trans}. See \textit{IsFunctionalDigraph} (6.1.9).

Specifically, the digraph returned by \textit{AsDigraph} has \textit{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 \textit{n} is not supplied, then the degree of the transformation \textit{trans} is used by default. If the restriction of \textit{trans} to \([1 \ldots n]\) does not define a transformation of \([1 \ldots n]\), then \textit{AsDigraph(trans, n)} returns \textit{fail}.

Example

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

3.2.3 Graph

```gap
> Graph(digraph)  \quad \text{(operation)}

\textbf{Returns:} A \textit{Grape} package graph.

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

If \textit{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 Graph(DigraphRemoveAllMultipleEdges(digraph)).

Example

```gap
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>
gap> Graph(last) = Petersen;
true
```
3.2.4 AsGraph

AsGraph(digraph)

Returns: A Grape package graph.

If digraph is a digraph, then this method returns the same as Graph (3.2.3), except that if digraph is immutable, then the result will be stored as a mutable attribute of digraph. In this latter case, when AsGraph(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> G := AsGraph(D);
rec( adjacencies := [ [ 1, 2 ], [ 3 ], [ ] ], group := Group(()),
    isGraph := true, names := [ 1 .. 3 ], order := 3,
    representatives := [ 1, 2, 3 ], schreierVector := [ -1, -2, -3 ] )

3.2.5 AsTransformation

AsTransformation(digraph)

Returns: A transformation, or fail

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

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

Example

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

3.3 New digraphs from old

3.3.1 DigraphImmutableCopy

DigraphImmutableCopy(digraph)

DigraphMutableCopy(digraph)

DigraphCopySameMutability(digraph)
Digraphs

\textbf{DigraphCopy(digraph)}

\textbf{Returns:} A digraph.

Each of these operations returns a new copy of \texttt{digraph}, of the appropriate mutability, retaining none of the attributes or properties of \texttt{digraph}.

\texttt{DigraphCopy} is a synonym for \texttt{DigraphCopySameMutability}.

\textbf{Example}

\begin{verbatim}
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>
\end{verbatim}

\textbf{3.3.2 DigraphImmutableCopyIfImmutable}

\textbf{DigraphImmutableCopyIfImmutable(digraph)}

\textbf{DigraphImmutableCopyIfMutable(digraph)}

\textbf{DigraphMutableCopyIfMutable(digraph)}

\textbf{DigraphMutableCopyIfImmutable(digraph)}

\textbf{Returns:} A digraph.

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

\textbf{Example}

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

3.3.3 InducedSubdigraph

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

*(operation)*

**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 \text{DigraphVertexLabels} (5.1.9).

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

**Example**

```
gap> D := Digraph([[1, 1, 2, 3, 4, 4], [1, 3, 4], [3, 1], [1, 1]]);
<immutable multidigraph with 4 vertices, 13 edges>
> InducedSubdigraph(D, [1, 3, 4]);
<immutable multidigraph with 3 vertices, 9 edges>
> DigraphVertices(last);
[ 1 .. 3 ]
> D := DigraphMutableCopy(D);
<mutable multidigraph with 4 vertices, 13 edges>
> new := InducedSubdigraph(D, [1, 3, 4]);
<mutable multidigraph with 3 vertices, 9 edges>
> D = new;
true
```
3.3.4 ReducedDigraph

\[ \text{ReducedDigraph}(\text{digraph}) \]
\[ \text{ReducedDigraphAttr}(\text{digraph}) \]

**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.

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

3.3.5 MaximalSymmetricSubdigraph

\[ \text{MaximalSymmetricSubdigraph}(\text{digraph}) \]
\[ \text{MaximalSymmetricSubdigraphAttr}(\text{digraph}) \]
\[ \text{MaximalSymmetricSubdigraphWithoutLoops}(\text{digraph}) \]
\[ \text{MaximalSymmetricSubdigraphWithoutLoopsAttr}(\text{digraph}) \]

**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 \( \text{digraph} \) 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
D := Digraph([[2, 2], [1, 3], [4], [3, 1]]);  # immutable multidigraph with 4 vertices, 7 edges
not IsSymmetricDigraph(D) and IsMultiDigraph(D);  # true
OutNeighbours(D);
[ [ 2, 2 ], [ 1, 3 ], [ 4 ], [ 3, 1 ] ]
S := MaximalSymmetricSubdigraph(D);  # immutable symmetric digraph with 4 vertices, 4 edges
IsSymmetricDigraph(S) and not IsMultiDigraph(S);  # true
OutNeighbours(S);
[ [ 2 ], [ 1 ], [ 4 ], [ 3 ] ]
D := CycleDigraph(IsMutableDigraph, 3);  # mutable digraph with 3 vertices, 3 edges
MaximalSymmetricSubdigraph(D);  # mutable empty digraph with 3 vertices
D;
# mutable empty digraph with 3 vertices
```

### 3.3.6 MaximalAntiSymmetricSubdigraph

> MaximalAntiSymmetricSubdigraph(digraph)  
> MaximalAntiSymmetricSubdigraphAttr(digraph)

**Returns:** A digraph.

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

If `digraph` is immutable, then a new immutable digraph is returned. Otherwise, the mutable digraph `digraph` is changed in-place.

See IsAntiSymmetricDigraph (6.1.2) for more information.

Example

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

<mutable digraph with 2 vertices, 1 edge>
gap> D;
<mutable digraph with 2 vertices, 1 edge>

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
<immutable multidigraph with 4 vertices, 9 edges>
gap> D := Digraph([[1, 2, 1, 3], [1], [4], [3, 4, 3]]);
<immutable symmetric digraph with 4 vertices, 4 edges>
gap> forest := UndirectedSpanningForest(D);
fail
gap> OutNeighbours(forest);
[ [ 2 ], [ 1 ], [ 4 ], [ 3 ] ]
gap> IsUndirectedSpanningForest(D, forest);
true
gap> DigraphConnectedComponents(forest).comps;
[ [ 1, 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);
<immutable symmetric digraph with 4 vertices, 6 edges>
gap> IsUndirectedSpanningTree(D, tree);
true
gap> tree = UndirectedSpanningForest(D);
true
gap> UndirectedSpanningForest(EmptyDigraph(0));
fail
```
3.3.8 QuotientDigraph

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

**Returns:** A digraph.

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

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

The vertices of \( \text{digraph} \) in part \( i \) of \( p \) will become vertex \( i \) in the quotient, and there exists some edge in \( \text{digraph} \) with source in part \( i \) and range in part \( j \) if and only if there is an edge from \( i \) to \( j \) 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 **Digraphs** package.

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

**Example**

```gap
gap> D := PetersenGraph(IsMutableDigraph);  
<mutable digraph with 10 vertices, 30 edges>
gap> UndirectedSpanningTree(D);  
<mutable digraph with 10 vertices, 18 edges>
gap> D;  
<mutable digraph with 10 vertices, 18 edges>
```

3.3.9 DigraphReverse

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

**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 \textit{digraph}. If \textit{digraph} is immutable, then a new immutable digraph is returned. Otherwise, the mutable digraph \textit{digraph} is changed in-place into its reverse.

\begin{verbatim}
Example
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> OutNeighbours(D);
[ [ 3 ], [ 1 ], [ 2 ] ]
gap> DigraphReverse(D);
<mutable digraph with 3 vertices, 3 edges>
gap> OutNeighbours(D);
[ [ 3 ], [ 1 ], [ 2 ] ]
\end{verbatim}

3.3.10 DigraphDual

\begin{verbatim}
Example
D := Digraph([[[2, 3], [], [4, 6], [5], []],
> [7, 8, 9], [], [], [], [1]]);
<immutable digraph with 9 vertices, 8 edges>
D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
D := Digraph([[[2, 3], [], [4, 6], [5], []],
> [7, 8, 9], [], [], [], [1]]);
<immutable digraph with 9 vertices, 73 edges>
D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 6 edges>
D;
<mutable digraph with 3 vertices, 6 edges>
\end{verbatim}
3.3.11 DigraphSymmetricClosure

- DigraphSymmetricClosure(digraph) (operation)
- DigraphSymmetricClosureAttr(digraph) (attribute)

**Returns:** A digraph.

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

A digraph is symmetric if its adjacency matrix 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 IsSymmetricDigraph (6.1.12).

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

```
Example

gap> D := Digraph([[1, 2, 3], [2, 4], [1], [3, 4]]);
<immutable digraph with 4 vertices, 8 edges>
gap> D := DigraphSymmetricClosure(D);
<immutable symmetric digraph with 4 vertices, 11 edges>
gap> IsSymmetricDigraph(D);
true
gap> List(OutNeighbours(D), AsSet);
[[1, 2, 3], [1, 4], [2, 3, 4]]
```

3.3.12 DigraphTransitiveClosure

- DigraphTransitiveClosure(digraph) (operation)
- DigraphTransitiveClosureAttr(digraph) (attribute)
- DigraphReflexiveTransitiveClosure(digraph) (operation)
- DigraphReflexiveTransitiveClosureAttr(digraph) (attribute)

**Returns:** A digraph.

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

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

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

Example

```gap
```
```
Digraphs

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

3.3.14 DigraphAddVertex

\texttt{DigraphAddVertex(digraph[, label])}

\textbf{Returns:} A digraph.

The operation returns a digraph constructed from \texttt{digraph} by adding a single new vertex, and no new edges.

If the optional second argument \texttt{label} is a \texttt{GAP} object, then the new vertex will be labelled \texttt{label}.

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

Example

\begin{verbatim}
gap> D := CompleteDigraph(3);
<immutable complete digraph with 3 vertices>
gap> new := DigraphAddVertex(D);
<immutable digraph with 4 vertices, 6 edges>
gap> D = new;
false
gap> DigraphVertices(new);
[ 1 .. 4 ]
gap> new := DigraphAddVertex(D, Group([[1, 2]]));
<immutable digraph with 4 vertices, 6 edges>
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>
\end{verbatim}
3.3.15 DigraphAddVertices (for a digraph and an integer)

\[ \text{DigraphAddVertices} \]

- \text{DigraphAddVertices}(\text{digraph}, m)
- \text{DigraphAddVertices}(\text{digraph}, \text{labels})

**Returns:** A digraph.

For a non-negative integer \( m \), this operation returns a digraph constructed from \text{digraph} by adding \( m \) new vertices.

Otherwise, if \text{labels} is a list consisting of \( k \) \text{GAP} objects, then this operation returns a digraph constructed from \text{digraph} by adding \( k \) new vertices, which are labelled according to this list.

If \text{digraph} belongs to \text{IsMutableDigraph} (3.1.2), then the vertices are added directly to \text{digraph}, which is changed in-place. If \text{digraph} belongs to \text{IsImmutableDigraph} (3.1.3), then \text{digraph} itself is returned if no vertices are added (i.e. \( m=0 \) or \text{labels} is empty), otherwise the result is a new immutable digraph.

**Example**

```gap
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)

\[ \text{DigraphAddEdge} \]

- \text{DigraphAddEdge}(\text{digraph}, \text{edge})
- \text{DigraphAddEdge}(\text{digraph}, \text{src}, \text{ran})

**Returns:** A digraph.

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

If \text{digraph} belongs to \text{IsMutableDigraph} (3.1.2), then the edge is added directly to \text{digraph}. If \text{digraph} belongs to \text{IsImmutableDigraph} (3.1.3), then an immutable copy of \text{digraph} with the additional edge is returned.
3.3.17 DigraphAddEdgeOrbit

\texttt{DigraphAddEdgeOrbit(digraph, edge)}

\textbf{Returns:} A new digraph.

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

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

\begin{example}
\begin{verbatim}
gap> gr1 := CayleyDigraph(DihedralGroup(8));
<immutable digraph with 8 vertices, 24 edges>
\end{verbatim}
\end{example}
3.3.18 DigraphAddEdges

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

Returns: A digraph.

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

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

Example

\begin{verbatim}
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> <immutable digraph with 1024 vertices, 1022 edges>
gap> new := DigraphAddEdges(D, 
gap>   [[1023, 1024], [1, 1024], [1023, 1024], [1024, 1]]);
gap> <immutable multidigraph with 1024 vertices, 1026 edges>
gap> D = new; false
gap> D2 := DigraphMutableCopy(func(1024));
gap> <mutable digraph with 1024 vertices, 1022 edges>
gap> new := DigraphAddEdges(D2, 
gap>   [[1023, 1024], [1, 1024], [1023, 1024], [1024, 1]]);
gap> <mutable multidigraph with 1024 vertices, 1026 edges>
gap> D2 = new; true
\end{verbatim}

3.3.19 DigraphRemoveVertex

\[ \text{DigraphRemoveVertex} \left( \text{digraph}, \text{v} \right) \] (operation)

Returns: A digraph.

If \( \text{v} \) is a vertex of \( \text{digraph} \), then this operation returns a digraph constructed from \( \text{digraph} \) by removing vertex \( \text{v} \), along with any edge whose source or range vertex is \( \text{v} \).

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

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

Example

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

### 3.3.20 DigraphRemoveVertices

**Example**

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

\[ \text{DigraphRemoveEdge}(\text{digraph}, \text{edge}) \] (operation)
\[ \text{DigraphRemoveEdge}(\text{digraph}, \text{src}, \text{ran}) \] (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]);;
true
gap> IsIdenticalObj(new, D);
true
gap> D := DigraphMutableCopy(D);;
true
```

### 3.3.22 DigraphRemoveEdgeOrbit

\[ \text{DigraphRemoveEdgeOrbit}(\text{digraph}, \text{edge}) \] (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> DigraghEdges(gr1);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 1 ], [ 2, 8 ], [ 2, 6 ],
 [ 3, 5 ], [ 3, 4 ], [ 3, 7 ], [ 4, 6 ], [ 4, 7 ], [ 4, 1 ],
 [ 5, 3 ], [ 5, 2 ], [ 5, 8 ], [ 6, 4 ], [ 6, 5 ], [ 6, 2 ],
 [ 7, 8 ], [ 7, 1 ], [ 7, 3 ], [ 8, 7 ], [ 8, 6 ], [ 8, 5 ] ]
gap> DigraghEdges(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 ],
```
Digraphs

\[ [4, 6], [4, 7], [4, 1], [4, 5], [5, 3], [5, 2], [5, 8], [5, 4], [6, 4], [6, 5], [6, 2], [6, 7], [7, 8], [7, 1], [7, 3], [7, 6], [8, 7], [8, 6], [8, 5], [8, 1] \]

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

3.3.23 DigraphRemoveEdges

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

\textbf{Returns:} A digraph.

If one of the following holds:

- \texttt{digraph} is a digraph with no multiple edges, and \texttt{edges} is a list of pairs of vertices of \texttt{digraph}, or
- \texttt{digraph} is a digraph and \texttt{edges} is an empty list

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

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

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

\textbf{Example}

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

3.3.24 DigraphRemoveLoops

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

\textbf{Returns:} A digraph.

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

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

\[ \text{DigraphRemoveAllMultipleEdges} \]
\[ \text{operation} \]
\[ \text{DigraphRemoveAllMultipleEdgesAttr} \]
\[ \text{attribute} \]

Returns:
A digraph.

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

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

Example

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

3.3.26 DigraphReverseEdges (for a digraph and a list of edges)

\[ \text{DigraphReverseEdges} \]
\[ \text{operation} \]
\[ \text{DigraphReverseEdge} \]
\[ \text{operation} \]
\[ \text{DigraphReverseEdge} \]
\[ \text{operation} \]

Returns:
A digraph.

If `digraph` is a digraph without multiple edges, and `edges` is a list of pairs of vertices of `digraph` (the entries of each pair corresponding to the source and the range of an edge, respectively), then `DigraphReverseEdges` returns a digraph constructed from `digraph` by reversing the orientation of every edge specified by `edges`. If only one edge is to be reversed, then `DigraphReverseEdge` can be used instead. In this case, the second argument should just be a single vertex-pair, or 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.

Example

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

Example

```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);
true
gap> D2 := DigraphReverseEdges(D2, [[19, 7]]);
true
gap> D2 = new;
true
```

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

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.

Example

```gap
gap> D1 := CycleDigraph(3);
<immutable cycle digraph with 3 vertices>
gap> OutNeighbours(D1);
[ [ 2 ], [ 3 ], [ 1 ] ]
gap> D2 := CompleteDigraph(3);
<immutable complete digraph with 3 vertices>
gap> OutNeighbours(D2);
[ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ]
```
3.3.28 DigraphEdgeUnion (for a positive number of digraphs)

Returns: A digraph.

In the first form, if \( D1, D2, \ldots \) are digraphs, then DigraphEdgeUnion returns their edge union. In the second form, if \( list \) is a non-empty list of digraphs, then DigraphEdgeUnion returns the edge union of the digraphs contained in the list.

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

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

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

Example

```gap
gap> union := DigraphDisjointUnion(D1, D2);
<immutable digraph with 6 vertices, 9 edges>
gap> OutNeighbours(union);
[ [ 2 ], [ 3 ], [ 1 ], [ 5, 6 ], [ 4, 6 ], [ 4, 5 ]] 
```

3.3.29 DigraphJoin (for a positive number of digraphs)

Returns: A digraph.

In the first form, if \( D1, D2, \ldots \) are digraphs, then DigraphJoin returns their join. In the second form, if \( list \) is a non-empty list of digraphs, then DigraphJoin returns the join of the digraphs contained in the list.

The join of a collection of digraphs \( D1, D2, \ldots \) is formed by first taking the 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 DigraphVertexLabels (5.1.9) will be lost.

If the first digraph $D1 [\text{list}[1]]$ belongs to IsMutableDigraph (3.1.2), then $D1 [\text{list}[1]]$ 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.

Example

```gap
gap> D := CompleteDigraph(3);
<immutable complete digraph with 3 vertices>
gap> IsCompleteDigraph(DigraphJoin(D, D));
true
gap> D2 := CycleDigraph(3);
<immutable cycle digraph with 3 vertices>
gap> DigraphJoin(D, D2);
<immutable digraph with 6 vertices, 27 edges>
```

3.3.30 DigraphCartesianProduct (for a positive number of digraphs)

- DigraphCartesianProduct(\text{gr1, gr2, ...})
- DigraphCartesianProduct(\text{list})

Returns: A digraph.

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

In the second form, if $\text{list}$ is a non-empty list of digraphs, then 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 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 DigraphCartesianProductProjections (3.3.32). In addition, 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. DigraphVertexLabel(D,i) = [DigraphVertexLabel(G,u), DigraphVertexLabel(H,u')]).

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

Example

```gap
gap> gr := ChainDigraph(4);
<immutable chain digraph with 4 vertices>
gap> gr2 := CycleDigraph(3);
<immutable cycle digraph with 3 vertices>
gap> gr3 := DigraphCartesianProduct(gr, gr2);
<immutable digraph with 12 vertices, 21 edges>
gap> IsIsomorphicDigraph(gr3, > Digraph([[2, 5], [3, 6], [4, 7], [8],
```
3.3.31 DigraphDirectProduct (for a positive number of digraphs)

DigraphDirectProduct\( (\text{gr1, gr2, ...}) \)  
(Math: DigraphDirectProduct\( (\text{list}) \))

**Returns:** A digraph.

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

In the second form, if \( \text{list} \) is a non-empty list of digraphs, then 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 DigraphDirectProduct 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 DigraphDirectProductProjections (3.3.33). In addition 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. DigraphVertexLabel\( (D,i) = [\text{DigraphVertexLabel(G,u), 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.

```
Example
```

```
gap> gr := ChainDigraph(4);
<immutable chain digraph with 4 vertices>
gap> gr2 := CycleDigraph(3);
<immutable cycle digraph with 3 vertices>
gap> gr3 := DigraphDirectProduct(gr, gr2);
<immutable digraph with 12 vertices, 9 edges>
gap> IsIsomorphicDigraph(gr3,
> Digraph([[6], [7], [8], []],
> [10], [11], [12], []],
> [2], [3], [4], []));
true
```

3.3.32 DigraphCartesianProductProjections

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

**Returns:** A list of transformations.

If \( \text{digraph} \) is a Cartesian product digraph, \( \text{digraph} = \text{DigraphCartesianProduct(gr_1, gr_2, ...}) \), then DigraphCartesianProductProjections returns a list \( \text{proj} \) such that \( \text{proj}[i] \) is the projection onto the \(i\)-th coordinate of the product.

A projection is an idempotent endomorphism of \( \text{digraph} \). If \( \text{gr1, gr2, ...} \) are all loopless digraphs, then the induced subdigraph of \( \text{digraph} \) on the image of \( \text{proj}[i] \) is isomorphic to \( \text{gr}_i \).
Currently this attribute is only set upon creating an immutable digraph via DigraphCartesianProduct and there is no way of calculating it for other digraphs.
For more information see DigraphCartesianProduct (3.3.30)

Example

```gap
D := DigraphCartesianProduct(ChainDigraph(3), CycleDigraph(4),
  Digraph([[2], [2]]));
HasDigraphCartesianProductProjections(D);
true
proj := DigraphCartesianProductProjections(D);; Length(proj);
3
IsIdempotent(proj[2]);
true
RankOfTransformation(proj[3]);
2
S := ImageSetOfTransformation(proj[2]);;
IsIsomorphicDigraph(CycleDigraph(4), InducedSubdigraph(D, S));
true
```

3.3.33 DigraphDirectProductProjections

```
> DigraphDirectProductProjections(digraph)  (attribute)

Returns: A list of transformations.
If digraph is a direct product digraph, digraph = DigraphDirectProduct(gr_1, gr_2, ...
), then DigraphDirectProductProjections returns a list proj such that proj[i] is the
projection onto the i-th coordinate of the product.
A projection is an idempotent endomorphism of digraph. If gr1, gr2, ... are all loopless
digraphs, then the image of digraph under proj[i] is isomorphic to gr_i.
Currently this attribute is only set upon creating an immutable digraph via
DigraphDirectProduct and there is no way of calculating it for other digraphs.
For more information, see DigraphDirectProduct (3.3.31)
```

Example

```gap
D := DigraphDirectProduct(ChainDigraph(3), CycleDigraph(4),
  Digraph([[2], [2]]));
HasDigraphDirectProductProjections(D);
true
proj := DigraphDirectProductProjections(D);; Length(proj);
3
IsIdempotent(proj[2]);
true
RankOfTransformation(proj[3]);
2
P := DigraphRemoveAllMultipleEdges(
  ReducedDigraph(OnDigraphs(D, proj[2])));
IsIsomorphicDigraph(CycleDigraph(4), P);
true
```

3.3.34 LineDigraph

```
> LineDigraph(digraph)  (operation)
> EdgeDigraph(digraph)  (operation)
```
**Returns:** A digraph.

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

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

Example
---
```
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**
- **EdgeUndirectedDigraph**

**Returns:** A digraph.

Given a symmetric digraph `digraph`, the operation returns the symmetric digraph obtained by associating a vertex with each edge of `digraph`, ignoring directions and multiplicities, 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.

Example
---
```
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**

**Returns:** A digraph.

Let `digraph` be a digraph with vertex set `V`. This function returns the double digraph of `digraph`. The vertex set of the double digraph is the 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.

Example
---
```
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

- **BipartiteDoubleDigraph**

**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.

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

### 3.3.38 DigraphAddAllLoops

\noindent\textbf{DigraphAddAllLoops(digraph)} \hspace{1cm} \textbf{(operation)}

\noindent\textbf{DigraphAddAllLoopsAttr(digraph)} \hspace{1cm} \textbf{(attribute)}

\noindent\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.

### Example
```gap
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> OutNeighbours(D);
[ [ 1, 2, 3 ], [ 1, 3, 2 ], [ 1, 3 ] ]
gap> D := CycleDigraph(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>
```

### 3.3.39 DistanceDigraph (for digraph and int)

\noindent\textbf{DistanceDigraph(digraph, i)} \hspace{1cm} \textbf{(operation)}

\noindent\textbf{DistanceDigraph(digraph, list)} \hspace{1cm} \textbf{(operation)}

\noindent\textbf{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 \textit{digraph}, with two vertices
being adjacent if and only if the distance between them in $\text{digraph}$ equals $i$ or is a number in $\text{list}$. See `DigraphShortestDistance (5.3.2).

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

\begin{verbatim}
gap> digraph := DigraphFromSparse6String(
    > ":[n?AL'BC_DeEF'GlAHdIJeGKcKL_@McDHfILaBJfHMjKM'");
<immutable digraph with 30 vertices, 90 edges>
gap> DistanceDigraph(digraph, 1);
<immutable digraph with 30 vertices, 90 edges>
gap> DistanceDigraph(digraph, [1, 2]);
<immutable digraph with 30 vertices, 270 edges>
\end{verbatim}

3.3.40 DigraphClosure

\begin{verbatim}
\textbf{DigraphClosure(}digraph, k\textbf{)}
\end{verbatim}

\textbf{Returns:} A digraph.

Given a symmetric loopless digraph with no multiple edges $\text{digraph}$, the $k$-\textit{closure} of $\text{digraph}$ is defined to be the unique smallest symmetric loopless digraph $C$ with no multiple edges on the vertices of $\text{digraph}$ that contains all the edges of $\text{digraph}$ 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 $\text{digraph}$.

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

3.3.41 DigraphMycielskian

\begin{verbatim}
\textbf{DigraphMycielskian(}digraph\textbf{)}
\end{verbatim}

\begin{verbatim}
\textbf{DigraphMycielskianAttr(}digraph\textbf{)}
\end{verbatim}

\textbf{Returns:} A digraph.

If $\text{digraph}$ is a symmetric digraph, then `DigraphMycielskian` returns the Mycielskian of $\text{digraph}$.

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

\begin{verbatim}
gap> D := CycleDigraph(2);
<immutable cycle digraph with 2 vertices>
gap> ChromaticNumber(D);
\end{verbatim}
2

\begin{verbatim}
    gap> D := DigraphMycielskian(D);
    <immutable digraph with 5 vertices, 10 edges>
    gap> ChromaticNumber(D);
    3
    gap> D := DigraphMycielskian(D);
    <immutable digraph with 11 vertices, 40 edges>
    gap> ChromaticNumber(D);
    4
    gap> D := CompleteBipartiteDigraph(IsMutable, 2, 3);
    <mutable digraph with 5 vertices, 12 edges>
    gap> DigraphMycielskian(D);
    <mutable digraph with 11 vertices, 46 edges>
    gap> D;
    <mutable digraph with 11 vertices, 46 edges>
\end{verbatim}

3.4 Random digraphs

3.4.1 RandomDigraph

\begin{verbatim}
\textbf{Operation: } RandomDigraph([filt, ]n[, p])

\textbf{Returns:} A digraph.

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.

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

If the optional second argument \texttt{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 \texttt{p} is not specified, then a random probability will be assumed (chosen with uniform probability).

\textbf{Example: }

\begin{verbatim}
    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>
\end{verbatim}
\end{verbatim}

3.4.2 RandomMultiDigraph

\begin{verbatim}
\textbf{Operation: } RandomMultiDigraph(n[, m])

\textbf{Returns:} A digraph.

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

\textbf{Example: }

\begin{verbatim}
    gap> RandomMultiDigraph(1000);
    <immutable digraph with 1000 vertices, 364444 edges>
    gap> RandomMultiDigraph(10000, 0.023);
    <immutable digraph with 10000 vertices, 2300438 edges>
    gap> RandomMultiDigraph(IsMutableDigraph, 1000, 1 / 2);
    <mutable digraph with 1000 vertices, 499739 edges>
\end{verbatim}
\end{verbatim}
The method used by this function chooses each edge from the set of all possible edges with uniform probability. No effort is made to avoid creating multiple edges, so it is possible (but not guaranteed) that the result will have multiple edges. The result may or may not have loops.

Example

\begin{verbatim}
gap> RandomMultiDigraph(1000);
<immutable multidigraph with 1000 vertices, 216659 edges>

gap> RandomMultiDigraph(1000, 950);
<immutable multidigraph with 1000 vertices, 950 edges>
\end{verbatim}

3.4.3 RandomTournament

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

\textbf{Returns:} A digraph.

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.

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

Example

\begin{verbatim}
gap> RandomTournament(10);
<immutable tournament with 10 vertices>

gap> RandomTournament(IsMutableDigraph, 10);
<mutable digraph with 1000 vertices, 500601 edges>
\end{verbatim}

3.4.4 RandomLattice

\begin{verbatim}
\texttt{RandomLattice(n)}
\end{verbatim}

\textbf{Returns:} A digraph.

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.

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

Example

\begin{verbatim}
gap> RandomLattice(10);
<immutable lattice digraph with 10 vertices, 39 edges>

gap> RandomLattice(IsMutableDigraph, 10);
<mutable digraph with 12 vertices, 52 edges>
\end{verbatim}
3.5 Standard examples

3.5.1 ChainDigraph

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

(operation)

**Returns:** A digraph.

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

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 \( \text{DigraphReflexiveTransitiveClosure} \) (3.3.12) of a chain represents a total order.

\[ \text{gap} > \text{ChainDigraph}(42); \]
\[ <\text{immutable chain digraph with 42 vertices}> \]
\[ \text{gap} > \text{ChainDigraph}(\text{IsMutableDigraph}, 10); \]
\[ <\text{mutable digraph with 10 vertices, 9 edges}> \]

3.5.2 CompleteDigraph

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

(operation)

**Returns:** A digraph.

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

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.

\[ \text{gap} > \text{CompleteDigraph}(20); \]
\[ <\text{immutable complete digraph with 20 vertices}> \]
\[ \text{gap} > \text{CompleteDigraph}(\text{IsMutableDigraph}, 10); \]
\[ <\text{mutable digraph with 10 vertices, 90 edges}> \]

3.5.3 CompleteBipartiteDigraph

\[ \text{CompleteBipartiteDigraph}([\text{filt}, \ m, \ n]) \]

(operation)

**Returns:** A digraph.

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

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

\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}

### 3.5.4 CompleteMultipartiteDigraph

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

### 3.5.5 CycleDigraph

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

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

\[\text{NullDigraph}([\text{filt}, n])\] (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 IsMutableDigraph (3.1.2), then the digraph being created will be mutable, if \( \text{filt} \) is IsImmutableDigraph (3.1.3), then the digraph will be immutable. If the optional first argument \( \text{filt} \) is not present, then IsImmutableDigraph (3.1.3) is used by default.

NullDigraph is a synonym for EmptyDigraph.

**Example**

\[\text{gap} > \text{EmptyDigraph}(20);\]
\(<\text{immutable empty digraph with 20 vertices}>\)

\[\text{gap} > \text{NullDigraph}(10);\]
\(<\text{immutable empty digraph with 10 vertices}>\)

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

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 IsMutableDigraph (3.1.2), then the digraph being created will be mutable, if \( \text{filt} \) is IsImmutableDigraph (3.1.3), then the digraph will be immutable. If the optional first argument \( \text{filt} \) is not present, then 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**

\[\text{gap} > \text{gr} := \text{JohnsonDigraph}(3, 1);\]
\(<\text{immutable symmetric digraph with 3 vertices, 6 edges}>\)

\[\text{gap} > \text{OutNeighbours(gr)};\]
\([ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ]\)

\[\text{gap} > \text{gr} := \text{JohnsonDigraph}(4, 2);\]
\(<\text{immutable symmetric digraph with 6 vertices, 24 edges}>\)

\[\text{gap} > \text{OutNeighbours(gr)};\]
\([ [ 2, 3, 4, 5 ], [ 1, 3, 4, 6 ], [ 1, 2, 5, 6 ], [ 1, 2, 5, 6 ], [ 1, 3, 4, 6 ], [ 2, 3, 4, 5 ] ]\)

\[\text{gap} > \text{JohnsonDigraph}(1, 0);\]
\(<\text{immutable empty digraph with 1 vertex}>\)

\[\text{gap} > \text{JohnsonDigraph}(\text{IsMutableDigraph}, 1, 0);\]
\(<\text{mutable empty digraph with 1 vertex}>\)
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
> 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>
> GeneralisedPetersenGraph(40, 1);
<immutable symmetric digraph with 80 vertices, 240 edges>
> D := GeneralisedPetersenGraph(5, 2);
<immutable symmetric digraph with 10 vertices, 30 edges>
> IsIsomorphicDigraph(D, PetersenGraph());
true
> 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

\( \triangleright \) 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

\[
\begin{array}{l}
gap> g := \text{Digraph([[2, 3], [1], [2, 3]]);}
gap> \text{<immutable digraph with 3 vertices, 5 edges>}
\end{array}
\]
Digraphs

52

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

gap> IsSubdigraph(h, g);
false

gap> IsSubdigraph(CycleDigraph(4), ChainDigraph(4));
true

gap> IsSubdigraph(CompleteDigraph(4), CycleDigraph(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

⊿

IsUndirectedSpanningTree(
super, sub
)
(operation)

⊿

IsUndirectedSpanningForest(
super, sub
)
(operation)

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 (6.3.9)). 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 (6.3.9)), 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.

Example

gap> D := CompleteDigraph(4);
<immutable complete digraph with 4 vertices>
gap> tree := Digraph([[3], [4], [1, 4], [2, 3]]);
<immutable digraph with 4 vertices, 6 edges>
gap> IsSubdigraph(D, tree) and IsUndirectedTree(tree);
true

gap> IsUndirectedSpanningTree(D, tree);
true

gap> forest := EmptyDigraph(4);
<immutable empty digraph with 4 vertices>
gap> IsSubdigraph(D, forest) and IsUndirectedForest(forest);
true
gap> IsUndirectedSpanningForest(D, forest); 
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

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

**Returns:** A list of integers.

Returns the vertices of the digraph `digraph`.

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

Example

\[
\text{gap> gr := Digraph(["a", "b", "c"],}
> \text{["a", "b", "b"],}
> \text{["b", "c", "a"]);}
\]

\text{<immutable digraph with 3 vertices, 3 edges>}

\[
\text{gap> DigraphVertices(gr);}
\]

\[ [ 1 .. 3 ] \]

\[
\text{gap> gr := Digraph([1, 2, 3, 4, 5, 7],}
> \text{[1, 2, 2, 4, 4],}
> \text{[2, 7, 5, 3, 7]);}
\]

\text{<immutable digraph with 6 vertices, 5 edges>}

\[
\text{gap> DigraphVertices(gr);}
\]

\[ [ 1 .. 6 ] \]

\[
\text{gap> DigraphVertices(RandomDigraph(100));}
\]

\[ [ 1 .. 100 ] \]

\[
\text{gap> D := CycleDigraph(IsMutableDigraph, 3);}
\]

\text{<mutable digraph with 3 vertices, 3 edges>}

\[
\text{gap> DigraphVertices(D);}
\]

\[ [ 1 .. 3 ] \]

5.1.2 DigraphNrVertices

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

**Returns:** An integer.

Returns the number of vertices of the digraph `digraph`.

Example

\[
\text{gap> gr := Digraph(["a", "b", "c"],}
> \text{["a", "b", "b"],}
> \text{["b", "c", "a"]);}
\]

\text{<immutable digraph with 3 vertices, 3 edges>}

\[
\text{gap> DigraphNrVertices(gr);}
\]

\[ 3 \]
Digraphs

5.1.3 DigraphEdges

DigraphEdges(digraph) (attribute)

Returns: A list of lists.

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

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

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

Example

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

5.1.4 DigraphNrEdges

DigraphNrEdges(digraph) (attribute)

Returns: An integer.

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

Example

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

5.1.5 DigraphSinks

\textbf{DigraphSinks}(\textit{digraph}) \hspace{1cm} (\textbf{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 \textbf{OutDegreeOfVertex} (5.2.10).

\begin{verbatim}
gap> gr := Digraph([[3, 5, 2, 2], [3], [1], [5, 2, 5, 3], [1]]); 
<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} (\textbf{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 \textbf{InDegreeOfVertex} (5.2.12).

\begin{verbatim}
gap> gr := Digraph([[3, 5, 2, 2], [3], [1], [5, 2, 5, 3], [1]]); 
<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} (\textbf{attribute})

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

If \textit{digraph} is a digraph whose only directed cycles are loops, then \textbf{DigraphTopologicalSort} returns the vertices of \textit{digraph} ordered so that every edge’s source appears no earlier in the list than
Digraphs

its range. If the digraph digraph contains directed cycles of length greater than 1, then this operation returns fail.

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

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

Example

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

5.1.8 DigraphVertexLabel

\[\text{DigraphVertexLabel(digraph, i)}\] (operation)

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

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

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

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

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

Example

\[
\text{gap> D := DigraphFromDigraph6String("&DHUEe_.");}
<\text{immutable digraph with 5 vertices, 11 edges}>
\text{gap> DigraphVertexLabel(D, 3);}\]
3
\text{gap> D := Digraph(["a", "b", "c"], [], []);}
<\text{mutable empty digraph with 3 vertices}>
\text{gap> DigraphVertexLabel(D, 2);}
"b"
\text{gap> SetDigraphVertexLabel(D, 2, "d");}
\text{gap> DigraphVertexLabel(D, 2);}
"d"
\text{gap> D := InducedSubdigraph(D, [1, 2]);}
<\text{mutable empty digraph with 2 vertices}>
\text{gap> DigraphVertexLabel(D, 2);}
"d"
\text{gap> D := Digraph(IsMutableDigraph, ["e", "f", "g"], [], []);}
<\text{mutable empty digraph with 3 vertices}>
\text{gap> DigraphVertexLabel(D, 1);}
"e"
\text{gap> SetDigraphVertexLabel(D, 1, "h");}
5.1.9 DigraphVertexLabels

\[ \text{DigraphVertexLabels(digraph)} \quad \text{(operation)} \]
\[ \text{SetDigraphVertexLabels(digraph, list)} \quad \text{(operation)} \]

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

\[ \text{gap> D := DigraphFromDigraph6String("&DHUEe_");} \]
\[ \text{<immutable digraph with 5 vertices, 11 edges> } \]
\[ \text{gap> DigraphVertexLabels(D); } \]
\[ \text{[1 .. 5]} \]
\[ \text{gap> D := Digraph(["a", "b", "c"], [], []); } \]
\[ \text{<immutable empty digraph with 3 vertices> } \]
\[ \text{gap> DigraphVertexLabels(D); } \]
\[ \text{["a", "b", "c"]} \]
\[ \text{gap> SetDigraphVertexLabel(D, 2, "d"); } \]
\[ \text{gap> DigraphVertexLabels(D); } \]
\[ \text{["a", "d", "c"]} \]
\[ \text{gap> D := InducedSubdigraph(D, [1, 3]); } \]
\[ \text{<immutable empty digraph with 2 vertices> } \]
\[ \text{gap> DigraphVertexLabels(D); } \]
\[ \text{["a", "c"]} \]
\[ \text{gap> D := Digraph(IsMutableDigraph, ["e", "f", "g"], [], []); } \]
\[ \text{<mutable empty digraph with 3 vertices> } \]
\[ \text{gap> SetDigraphVertexLabels(D, ["h", "i", "j"]); } \]
\[ \text{gap> DigraphVertexLabels(D); } \]
\[ \text{["h", "i", "j"]} \]
\[ \text{gap> InducedSubdigraph(D, [1, 3]); } \]
\[ \text{<mutable empty digraph with 2 vertices> } \]
\[ \text{gap> DigraphVertexLabels(D); } \]
\[ \text{["h", "j"]} \]
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> 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 ] ]
gap> D := CompleteBipartiteDigraph(IsMutableDigraph, 2, 3);
<mutable digraph with 5 vertices, 12 edges>
gap> DigraphEdgeLabels(D);
[ [ 1, 1, 1 ], [ 1, 1, 1 ], [ 1, 1 ], [ 1, 1 ], [ 1, 1 ] ]
gap> SetDigraphEdgeLabel(D, 2, 4, "a");
gap> DigraphEdgeLabels(D);
[ [ 1, 1, 1 ], [ 1, "a", 1 ], [ 1, 1 ], [ 1, 1 ], [ 1, 1 ] ]
gap> InducedSubdigraph(D, [1, 2, 3, 4]);
<mutable digraph with 4 vertices, 8 edges>
gap> DigraphEdgeLabels(D);
[ [ 1, 1 ], [ 1, "a" ], [ 1, 1 ], [ 1, 1 ] ]
gap> OutNeighbors(D);
[ [ 3, 4 ], [ 3, 4 ], [ 1, 2 ], [ 1, 2 ] ]
```

5.1.12 **DigraphInEdges**

> DigraphInEdges(digraph, vertex)  (operation)

**Returns:** A list of edges.

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

Example

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

5.1.13 **DigraphOutEdges**

> DigraphOutEdges(digraph, vertex)  (operation)

**Returns:** A list of edges.

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

Example

```
gap> 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)

\[ \text{IsDigraphEdge}(\text{digraph}, \text{list}) \quad \text{(operation)} \]

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

Returns: true or false.

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

The second form simply returns true if \([u, v] \) is an edge in \text{digraph}, and false otherwise.

Example

\[ \text{gap} > \text{D} := \text{Digraph}([[2, 2], [6], [], [3], [], [1]]); \]
<immutable multidigraph with 6 vertices, 5 edges>
\[ \text{gap} > \text{IsDigraphEdge}(\text{D}, [1, 1]); \]
false
\[ \text{gap} > \text{IsDigraphEdge}(\text{D}, [1, 2]); \]
true
\[ \text{gap} > \text{IsDigraphEdge}(\text{D}, [1, 8]); \]
false

5.1.15 IsMatching

\[ \text{IsMatching}(\text{digraph}, \text{list}) \quad \text{(operation)} \]

\[ \text{IsMaximalMatching}(\text{digraph}, \text{list}) \quad \text{(operation)} \]

\[ \text{IsMaximumMatching}(\text{digraph}, \text{list}) \quad \text{(operation)} \]

\[ \text{IsPerfectMatching}(\text{digraph}, \text{list}) \quad \text{(operation)} \]

Returns: true or false.

If \text{digraph} is a digraph and \text{list} is a list of pairs of vertices of \text{digraph}, then \text{IsMatching} returns true if \text{list} is a matching of \text{digraph}. The operation \text{IsMaximalMatching} returns true if \text{list} is a maximal matching, \text{IsMaximumMatching} returns true if \text{list} is a maximum matching and \text{IsPerfectMatching} returns true if \text{list} is a perfect, matching of \text{digraph}, respectively. Otherwise, each of these operations return false.

A matching \( M \) of a digraph \text{digraph} is a subset of the edges of \text{digraph}, i.e. \text{DigraphEdges}(\text{digraph}), such that no pair of distinct edges in \( M \) are incident to the same vertex of \text{digraph}. Note that this definition allows a matching to contain loops. See \text{DigraphHasLoops} (6.1.1). The matching \( M \) is maximal if it is contained in no larger matching of the digraph, is maximum if it has the greatest cardinality among all matchings and is perfect if every vertex of the digraph is incident to an edge in the matching. Every maximum or perfect matching is maximal. Note, however, that not every perfect matching of digraphs with loops is maximum.

Example

\[ \text{gap} > \text{D} := \text{Digraph}([[1, 2], [1, 2], [2, 3, 4], [3, 5], [1]]); \]
<immutable digraph with 5 vertices, 10 edges>
\[ \text{gap} > \text{IsMatching}(\text{D}, [[2, 1], [3, 2]]); \]
false
\[ \text{gap} > \text{edges} := [[3, 2]]; \]
\[ \text{gap} > \text{IsMatching}(\text{D}, \text{edges}); \]
true
gap> IsMaximalMatching(D, edges);
false
gap> edges := [[2, 1], [3, 4]];;
gap> IsMaximalMatching(D, edges);
true
gap> IsPerfectMatching(D, edges);
false
gap> edges := [[1, 2], [3, 3], [4, 5]];;
gap> IsPerfectMatching(D, edges);
true
gap> edges := [[1, 1], [2, 2], [3, 3], [4, 5]];;
gap> IsMaximumMatching(D, edges);
false
gap> edges := [[1, 1], [2, 2], [3, 3], [4, 5]];;
gap> IsMaximumMatching(D, edges);
true

5.1.16 DigraphMaximalMatching

DigraphMaximalMatching(digraph)

Returns: A list of pairs of vertices.

This function returns a maximal matching of the digraph digraph.
For the definition of a maximal matching, see IsMaximalMatching (5.1.15).

Example

```gap
gap> D := DigraphFromDiSparse6String(".IeAoXCJU@|SHAe?d");
<immutable digraph with 10 vertices, 13 edges>
gap> M := DigraphMaximalMatching(D);; IsMaximalMatching(D, M);
true
gap> D := RandomDigraph(100);;
true
gap> IsMaximalMatching(D, DigraphMaximalMatching(D));
true
gap> D := GeneralisedPetersenGraph(IsMutableDigraph, 9, 2);
<mutable digraph with 18 vertices, 54 edges>
gap> IsMaximalMatching(D, DigraphMaximalMatching(D));
true
```

5.1.17 DigraphMaximumMatching

DigraphMaximumMatching(digraph)

Returns: A list of pairs of vertices.

This function returns a maximum matching of the digraph digraph.
For the definition of a maximum matching, see IsMaximumMatching (5.1.15). If digraph is bipartite (see IsBipartiteDigraph (6.1.3)), then the algorithm used has complexity \(O(m*\sqrt{n})\). Otherwise for general graphs the complexity is \(O(m*n*\log(n))\). Here \(n\) is the number of vertices and \(m\) is the number of edges.

Example

```gap
gap> D := DigraphFromDigraph6String("&I@EA_A?AdDp[_c??OO");
<immutable digraph with 10 vertices, 23 edges>
gap> M := DigraphMaximumMatching(D);; IsMaximalMatching(D, M);
true
```
Digraphs

gap> Length(M);
5

gap> D := Digraph([ [5, 6, 7, 8], [6, 7, 8], [7, 8], [8], [5, 7], [2, 6], [3, 7], [4, 8] ]);;

gap> M := DigraphMaximumMatching(D);
[ [ 1, 5 ], [ 2, 6 ], [ 3, 7 ], [ 4, 8 ] ]

gap> IsMaximalMatching(D, M);
true

gap> Length(M);
9

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

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

Example

| gap> gr := Digraph([ |
| > [2, 2, 2], [1, 3, 6, 8, 9, 10], [4, 6, 8], |
| > [1, 2, 3, 9], [3, 3], [3, 5, 6, 10], [1, 2, 7], |
| > [1, 2, 3, 10, 5, 6, 10], [1, 3, 4, 5, 8, 10], |
| > [2, 3, 4, 6, 7, 10]); |
| > <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, 0, 1, 1, 1 ], |
| [ 0, 0, 0, 1, 0, 1, 0, 1, 0 ], |
| [ 1, 1, 0, 0, 0, 0, 0, 1, 0 ], |
| [ 0, 0, 2, 0, 0, 0, 0, 0, 0 ], |
| [ 0, 0, 1, 0, 1, 0, 0, 0, 1 ], |
| [ 1, 1, 0, 0, 0, 0, 1, 0, 0 ], |
| [ 1, 1, 1, 0, 1, 1, 0, 0, 2 ], |
| [ 1, 0, 1, 1, 0, 0, 1, 0, 1 ] ] |

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

gap> Display(AdjacencyMatrix(D));
[ [ 0, 1, 0 ],
| [ 1, 0, 1 ], |
| [ 0, 0, 1 ] ] |

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

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

Example

| gap> gr := Digraph([ |
| > [2, 2, 2], [1, 3, 6, 8, 9, 10], [4, 6, 8], |
| > [1, 2, 3, 9], [3, 3], [3, 5, 6, 10], [1, 2, 7], |
| > [1, 2, 3, 10, 5, 6, 10], [1, 3, 4, 5, 8, 10], |
| > [2, 3, 4, 6, 7, 10]); |
| > <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, 0, 1, 1, 1 ], |
| [ 0, 0, 0, 1, 0, 1, 0, 1, 0 ], |
| [ 1, 1, 0, 0, 0, 0, 0, 1, 0 ], |
| [ 0, 0, 2, 0, 0, 0, 0, 0, 0 ], |
| [ 0, 0, 1, 0, 1, 0, 0, 0, 1 ], |
| [ 1, 1, 0, 0, 0, 0, 1, 0, 0 ], |
| [ 1, 1, 1, 0, 1, 1, 0, 0, 2 ], |
| [ 1, 0, 1, 1, 0, 0, 1, 0, 1 ] ] |

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

gap> Display(AdjacencyMatrix(D));
[ [ 0, 1, 0 ],
| [ 1, 0, 1 ], |
| [ 0, 0, 1 ] ] |
5.2.2 CharacteristicPolynomial

$\textbf{CharacteristicPolynomial}(\text{digraph})$

\textbf{Returns:} A polynomial with integer coefficients.

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

Example

```gap
gap> D := Digraph([2, 2, 2], [1, 3, 6, 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);
\_1^10-3\_1^9-7\_1^8-\_1^7+14\_1^6+\_1^5-26\_1^4+51\_1^3-10\_1^2
+18\_1-30
```

```gap
gap> D := CompleteDigraph(5);
<immutable complete digraph with 5 vertices>
```

```gap
gap> CharacteristicPolynomial(D);
\_1^5-10\_1^3-20\_1^2-15\_1-4
```

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

```gap
gap> CharacteristicPolynomial(D);
x_1^3-1
```

5.2.3 BooleanAdjacencyMatrix

$\textbf{BooleanAdjacencyMatrix}(\text{digraph})$

\textbf{Returns:} A square matrix of booleans.

If $\text{digraph}$ is a digraph with a positive number of vertices $n$, then $\text{BooleanAdjacencyMatrix}(\text{digraph})$ returns the boolean adjacency matrix $\text{mat}$ of $\text{digraph}$. The value of the matrix entry $\text{mat}[j][i]$ is true if and only if there exists an edge in $\text{digraph}$ with source $j$ and range $i$. If $\text{digraph}$ has no vertices, then the empty list is returned.

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

The attribute $\text{BooleanAdjacencyMatrix}$ returns an immutable list of immutable lists, whereas the function $\text{BooleanAdjacencyMatrixMutableCopy}$ returns a copy of the BooleanAdjacencyMatrix that is a mutable list of mutable lists.

Example

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

```gap
gap> PrintArray(BooleanAdjacencyMatrix(gr));
[ [ false, false, true, true ],
  [ false, true, true, false ],
  [ true, true, false, true ],
  [ false, false, false, true ]
]```
Digraphs

\[
\text{gap> gr := CycleDigraph(4);;}
\text{gap> PrintArray(BooleanAdjacencyMatrix(gr));}
\begin{array}{cccc}
\text{false} & \text{true} & \text{false} & \text{false} \\
\text{false} & \text{false} & \text{true} & \text{false} \\
\text{false} & \text{false} & \text{false} & \text{true} \\
\text{true} & \text{false} & \text{false} & \text{false}
\end{array}
\]
\text{gap> BooleanAdjacencyMatrix(EmptyDigraph(0));}
\[
\begin{array}{c}
\text{false}
\end{array}
\]
\text{gap> D := CycleDigraph(IsMutableDigraph, 3);}
\text{<mutable digraph with 3 vertices, 3 edges>}
\text{gap> PrintArray(BooleanAdjacencyMatrix(D));}
\begin{array}{ccc}
\text{false} & \text{true} & \text{false} \\
\text{false} & \text{false} & \text{true} \\
\text{true} & \text{false} & \text{false}
\end{array}
\]

5.2.4 DigraphAdjacencyFunction

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

\text{Returns: A function.}

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

\text{Example}

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

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

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

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

Example

\[
\text{gap> gr := Digraph([[2, 1, 3, 5], [1, 3, 4], [2, 3], [2], [1, 2, 3, 4]]); }
<\text{immutable digraph with 5 vertices, 14 edges}>
\]
\[
\text{gap> DigraphRange(gr); }
\text{[ 2, 1, 3, 5, 1, 3, 4, 2, 3, 2, 1, 2, 3, 4 ]}
\]
\[
\text{gap> DigraphSource(gr); }
\text{[ 1, 1, 1, 1, 2, 2, 2, 3, 3, 4, 5, 5, 5, 5 ]}
\]
\[
\text{gap> DigraphEdges(gr);} \quad \text{[ [ 1, 2 ], [ 1, 1 ], [ 1, 3 ], [ 1, 5 ], [ 2, 1 ], [ 2, 3 ], [ 2, 4 ], [ 3, 2 ], [ 3, 3 ], [ 4, 2 ], [ 5, 1 ], [ 5, 2 ], [ 5, 3 ], [ 5, 4 ] ]}
\]
\[
\text{gap> D := CycleDigraph(IsMutableDigraph, 3);} \quad <\text{mutable digraph with 3 vertices, 3 edges}>
\]
\[
\text{gap> DigraphRange(D); }
\text{[ 2, 3, 1 ]}
\]
\[
\text{gap> DigraphSource(D); }
\text{[ 1, 2, 3 ]}
\]

5.2.6 OutNeighbours

\[
\text{OutNeighbours}(\text{digraph}) \quad \text{(attribute)}
\]
\[
\text{OutNeighbors}(\text{digraph}) \quad \text{(attribute)}
\]
\[
\text{OutNeighboursMutableCopy}(\text{digraph}) \quad \text{(operation)}
\]
\[
\text{OutNeighborsMutableCopy}(\text{digraph}) \quad \text{(operation)}
\]

**Returns:** The adjacencies of a digraph.

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

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

\[
\text{gap> gr := Digraph(["a", "b", "c"], }
\text{ > ["a", "b", "b"], }
\text{ > ["b", "a", "c"];}
<\text{immutable digraph with 3 vertices, 3 edges}>
\]
\[
\text{gap> OutNeighbours(gr);} \quad \text{[ [ 1 ], [ 2 ], [ 3 ] ]}
\]
Digraphs

Digraphs

\[
\begin{bmatrix}
2, 1, 3 \\
1, 2, 3 \\
2, 1
\end{bmatrix}
\]

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

\[
\text{gap> OutNeighbours(gr);}
\]
\[
\begin{bmatrix}
1, 2, 3 \\
1, 2 \\
2
\end{bmatrix}
\]

\[
\text{gap> gr := DigraphByAdjacencyMatrix([}
\>
\>
\>
\]
\[
\text{gap> OutNeighbours(gr);}
\]
\[
\begin{bmatrix}
1, 2, 3 \\
1, 2 \\
2 
\end{bmatrix}
\]

\[
\text{gap> D := CycleDigraph(IsMutableDigraph, 3);}
\]
\[
<\text{mutable digraph with 3 vertices, 3 edges}>
\]

\[
\text{gap> OutNeighbours(D);}
\]
\[
\begin{bmatrix}
2 \\
3 \\
1
\end{bmatrix}
\]

5.2.7 InNeighbours

\[
\text{InNeighbours(digraph)} \quad \text{(attribute)}
\]
\[
\text{InNeighbors(digraph)} \quad \text{(attribute)}
\]
\[
\text{InNeighboursMutableCopy(digraph)} \quad \text{(operation)}
\]
\[
\text{InNeighborsMutableCopy(digraph)} \quad \text{(operation)}
\]

Returns: A list of lists of vertices.

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

The function \( \text{InNeighbours} \) returns an immutable list of lists, whereas the function \( \text{InNeighboursMutableCopy} \) returns a copy of \( \text{InNeighbours} \) which is a mutable list of mutable lists.

Note that each entry of \( \text{inn} \) is sorted into ascending order.

Example

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

\[
\text{gap> InNeighbours(gr);}
\]
\[
\begin{bmatrix}
1 \\
1 \\
1
\end{bmatrix}
\]

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

\[
\text{gap> InNeighbours(gr);}
\]
\[
\begin{bmatrix}
1, 2 \\
1, 2 \\
1
\end{bmatrix}
\]

\[
\text{gap> gr := DigraphByAdjacencyMatrix([}
\>
\>
\>
\]
\[
\text{gap> InNeighbours(gr);}
\]
\[
\begin{bmatrix}
1, 2 \\
1, 2 \\
1
\end{bmatrix}
\]
5.2.8 OutDegrees

$\text{OutDegrees(} \text{digraph} \text{)}$  \hspace{1cm} (attribute)  
$\text{OutDegreeSequence(} \text{digraph} \text{)}$  \hspace{1cm} (attribute)  
$\text{OutDegreeSet(} \text{digraph} \text{)}$  \hspace{1cm} (attribute)

\textbf{Returns:} A list of non-negative integers.  
Given a digraph $\text{digraph}$ with $n$ vertices, the function $\text{OutDegrees}$ returns an immutable list out of length $n$, such that for a vertex $i$ in $\text{digraph}$, the value of $\text{out}[i]$ is the out-degree of vertex $i$. See $\text{OutDegreeOfVertex}$ (5.2.10).  
The function $\text{OutDegreeSequence}$ returns the same list, after it has been sorted into non-increasing order.  
The function $\text{OutDegreeSet}$ returns the same list, sorted into increasing order with duplicate entries removed.

\textbf{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

$\text{InDegrees(} \text{digraph} \text{)}$  \hspace{1cm} (attribute)  
$\text{InDegreeSequence(} \text{digraph} \text{)}$  \hspace{1cm} (attribute)  
$\text{InDegreeSet(} \text{digraph} \text{)}$  \hspace{1cm} (attribute)

\textbf{Returns:} A list of non-negative integers.  
Given a digraph $\text{digraph}$ with $n$ vertices, the function $\text{InDegrees}$ returns an immutable list $\text{inn}$ of length $n$, such that for a vertex $i$ in $\text{digraph}$, the value of $\text{inn}[i]$ is the in-degree of vertex $i$. See $\text{InDegreeOfVertex}$ (5.2.12).  
The function $\text{InDegreeSequence}$ returns the same list, after it has been sorted into non-increasing order.  
The function $\text{InDegreeSet}$ returns the same list, sorted into increasing order with duplicate entries removed.
Example

```gap
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 ]
gap> InDegreeSequence(D);
[ 3, 2, 2, 1 ]
gap> InDegreeSet(D);
[ 1, 2, 3 ]
gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
gap> InDegrees(D);
[ 1, 1, 1 ]
```

5.2.10 OutDegreeOfVertex

▷ OutDegreeOfVertex(digraph, vertex)  

Returns: The non-negative integer.

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

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

5.2.11 OutNeighboursOfVertex

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

Returns: A list of vertices.

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

```gap
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> OutNeighboursOfVertex(D, 1);
[ 2, 2, 3 ]
gap> OutNeighboursOfVertex(D, 3);
[ 2, 2, 3 ]
```
5.2.12 InDegreeOfVertex

\> InDegreeOfVertex(digraph, vertex)  
(operation)

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

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

```
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
```

5.2.13 InNeighboursOfVertex

\> InNeighboursOfVertex(digraph, vertex)  
InNeighborsOfVertex(digraph, vertex)  
(operation)

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

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

```
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);  
[ 2, 4, 4, 4 ]
gap> InNeighboursOfVertex(D, 3);  
[ 1, 1, 3, 3, 4, 4, 4, 4 ]
gap> InNeighboursOfVertex(D, 4);  
[ ]
```

5.2.14 DigraphLoops

\> DigraphLoops(digraph)  
(attribute)

**Returns:** A list of vertices.

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

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

5.2.15 PartialOrderDigraphMeetOfVertices

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

Returns:
A positive integer or fail

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

Example

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

5.2.16 DegreeMatrix

▷ DegreeMatrix(digraph) (attribute)

Returns:
A square matrix of non-negative integers.

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

See OutDegrees (5.2.8) for more information.
Digraphs

Example

gap> D := Digraph([[1, 2, 3], [4], [1, 3, 4], []]);
<immutable digraph with 4 vertices, 7 edges>
gap> PrintArray(DegreeMatrix(D));
[ [ 3, 0, 0, 0 ],
  [ 0, 1, 0, 0 ],
  [ 0, 0, 3, 0 ],
  [ 0, 0, 0, 0 ] ]
gap> D := CycleDigraph(5);;
gap> PrintArray(DegreeMatrix(D));
[ [ 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 ] ]
gap> DegreeMatrix(EmptyDigraph(0));
[ ]
gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
gap> Display(DegreeMatrix(D));
[ [ 1, 0, 0 ],
  [ 0, 1, 0 ],
  [ 0, 0, 1 ] ]

5.2.17 LaplacianMatrix

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

\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 \text{DegreeMatrix(digraph)} - \text{AdjacencyMatrix(digraph)}. If \text{digraph} has no vertices, then the empty list is returned.

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

Example

gap> gr := Digraph([[1, 2, 3], [4], [1, 3, 4], []]);
<immutable digraph with 4 vertices, 7 edges>
gap> PrintArray(LaplacianMatrix(gr));
[ [ 2, -1, -1, 0 ],
  [ 0, 1, 0, -1 ],
  [ -1, 0, 2, -1 ],
  [ 0, 0, 0, 0 ] ]
gap> LaplacianMatrix(EmptyDigraph(0));
[ ]
gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
gap> Display(LaplacianMatrix(D));
[ [ 1, -1, 0 ],
  [ 0, 1, -1 ],
  [ -1, 0, 1 ] ]
5.3 Reachability and connectivity

5.3.1 DigraphDiameter

\begin{verbatim}
DigraphDiameter(digraph) (attribute)
\end{verbatim}

Returns: An integer or fail.

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

If a digraph \textit{digraph} is strongly connected and has at least 1 vertex, then the \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 fail.

See DigraphShortestDistances (5.3.3).

Example

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

5.3.2 DigraphShortestDistance (for a digraph and two vertices)

\begin{verbatim}
DigraphShortestDistance(digraph, u, v) (operation)  
DigraphShortestDistance(digraph, list) (operation)  
DigraphShortestDistance(digraph, list1, list2) (operation)
\end{verbatim}

Returns: An integer or 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 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.

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]);
\end{verbatim}
Digraphs

5.3.3 DigraphShortestDistances

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

\textbf{Returns:} A square matrix.

If \textit{digraph} is a digraph with \( n \) vertices, then this function returns an \( n \times n \) matrix \texttt{mat}, where each entry is either a non-negative integer, or \texttt{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 \texttt{mat}[i][j] is the length of the shortest such directed path. If no such directed path exists, then the value of \texttt{mat}[i][j] is \texttt{fail}. We use the convention that the distance from every vertex to itself is 0, i.e. \texttt{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 \( O(n^3) \).

\begin{verbatim}
Example
\end{verbatim}

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

5.3.4 DigraphLongestDistanceFromVertex

\textbf{DigraphLongestDistanceFromVertex}(\textit{digraph, v})

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

If \textit{digraph} is a digraph and \textit{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 \textit{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 \textit{v} which traverses a loop or a directed cycle, then we consider there to be a walk of infinite length from \textit{v} (realised by repeatedly traversing the loop/directed cycle), and so the result is infinity. To disallow walks using loops, try using \texttt{DigraphRemoveLoops (3.3.24)}:

\begin{verbatim}
DigraphLongestDistanceFromVertex(DigraphRemoveLoops(digraph, v)).
\end{verbatim}
• Otherwise, if all directed walks starting at vertex \( v \) have finite length, then the length of the longest such walk is returned.

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

Example

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

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

- DigraphDistanceSet(digraph, vertex, distance) (operation)
- DigraphDistanceSet(digraph, vertex, distances) (operation)

**Returns:** A list of vertices

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

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

Example

```gap
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 ]
```

5.3.6 DigraphGirth

- DigraphGirth(digraph) (attribute)

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

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

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

In the worst case, the method used in this function is a version of the Floyd-Warshall algorithm, and has complexity \( O(n^3) \), where \( n \) is the number of vertices in `digraph`. If the digraph has known automorphisms [see DigraphGroup (7.2.10)], then the performance is likely to be better.

For symmetric digraphs, see also DigraphUndirectedGirth (5.3.8).
### Example

```gap
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);
3

gap> D := GeneralisedPetersenGraph(IsMutableDigraph, 6, 2);
<mutable digraph with 12 vertices, 36 edges>
gap> DigraphGirth(D);
2

5.3.7 DigraphOddGirth

> DigraphOddGirth(digraph)                          (attribute)

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

This attribute returns the odd girth of the digraph `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 `digraph` has no directed cycles of odd length, then this function will return `infinity`, even if `digraph` has a directed cycle of even length. If `digraph` contains a loop, then this function will return 1.

See also `DigraphGirth (5.3.6)`.

```gap
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);
infinity
gap> D := Digraph([[2], [3], [], [3], [4]]);
<immutable digraph with 5 vertices, 4 edges>
gap> DigraphOddGirth(D);
infinity
gap> D := CycleDigraph(IsMutableDigraph, 4);
<mutable digraph with 4 vertices, 4 edges>
gap> DigraphDisjointUnion(D, CycleDigraph(5));
<mutable digraph with 9 vertices, 9 edges>
gap> DigraphOddGirth(D);
5
```
5.3.8 DigraphUndirectedGirth

> DigraphUndirectedGirth(digraph)  

Returns: An integer or infinity.

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

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

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

Example

```gap
gap> D := Digraph([[2, 4], [1, 3], [2, 4], [1, 3]]);
<immutable digraph with 4 vertices, 8 edges>
gap> DigraphUndirectedGirth(D);
4
gap> D := Digraph([[2], [1, 3], [2]]);
<immutable digraph with 3 vertices, 4 edges>
gap> DigraphUndirectedGirth(D);
infinity
gap> D := Digraph([[1], [], [4], [3]]);
<immutable digraph with 4 vertices, 3 edges>
gap> DigraphUndirectedGirth(D);
1
gap> D := GeneralisedPetersenGraph(IsMutableDigraph, 9, 2);
<mutable digraph with 18 vertices, 54 edges>
gap> DigraphUndirectedGirth(D);
5
```

5.3.9 DigraphConnectedComponents

> DigraphConnectedComponents(digraph)  

> DigraphNrConnectedComponents(digraph)  

Returns: A record.

This function returns the record \(wcc\) corresponding to the weakly connected components of the digraph \(digraph\). Two vertices of \(digraph\) are in the same weakly connected component whenever they are equal, or there exists a directed path (ignoring the orientation of edges) between them. More formally, two vertices are in the same weakly connected component of \(digraph\) if and only if they are in the same strongly connected component (see DigraphStronglyConnectedComponents (5.3.11)) 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.
5.3.10 DigraphConnectedComponent

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

(\text{operation})

**Returns:** A list of vertices.

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

Example

\[
\begin{align*}
gap> \text{D} & := \text{Digraph([[[3],[2],[1,2],[4]]]);} \\
& <\text{immutable digraph with 4 vertices, 5 edges}> \\
gap> \text{DigraphConnectedComponent}(\text{D}, 3); \\
& [1, 2, 3] \\
gap> \text{DigraphConnectedComponent}(\text{D}, 2); \\
& [1, 2, 3] \\
gap> \text{DigraphConnectedComponent}(\text{D}, 4); \\
& [4]
\end{align*}
\]

5.3.11 DigraphStronglyConnectedComponents

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

(\text{attribute})

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

(\text{attribute})

**Returns:** A record.

This function returns the record \text{scc} corresponding to the strongly connected components of the digraph \text{digraph}. Two vertices of \text{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 \text{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
gap> gr := Digraph([[2], [3, 1], []]);
<immutable digraph with 3 vertices, 3 edges>
gap> DigraphStronglyConnectedComponents(gr);
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);
<mutable digraph with 2 vertices, 2 edges>
gap> G := CycleDigraph(3);
<immutable cycle digraph with 3 vertices>
gap> D := 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
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

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

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

If \text{digraph} is a bipartite digraph, i.e. if it satisfies IsBipartiteDigraph (6.1.3), then \text{DigraphBicomponents} returns a pair of bicomponents of \text{digraph}. Otherwise, \text{DigraphBicomponents} returns \textit{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 \textit{bicomponents} of \text{digraph}. Equivalently, a pair of bicomponents of \text{digraph} consists of the color-classes of a 2-coloring of \text{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.

\begin{verbatim}
gap> D := CycleDigraph(3);
<immutable cycle digraph with 3 vertices>
gap> DigraphBicomponents(D);
fail

gap> D := ChainDigraph(5);
<immutable chain digraph with 5 vertices>
gap> DigraphBicomponents(D);
[ [ 1, 3, 5 ], [ 2, 4 ] ]

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

gap> D := CompleteBipartiteDigraph(IsMutableDigraph, 2, 3);
<mutable digraph with 5 vertices, 12 edges>
gap> DigraphBicomponents(D);
[ [ 1, 2 ], [ 3, 4, 5 ] ]
\end{verbatim}

5.3.14 ArticulationPoints

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

\textbf{Returns:} A list of vertices.

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

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

Multiple edges are ignored by this method.

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

If \( D \) has a bridge (see Bridges (5.3.15)), then a node incident to the bridge is an articulation point if and only if it has degree at least 2. It follows that if \( D \) has a bridge and at least 3 nodes, then at least one of the nodes incident to the bridge is an articulation point. The converse does not hold, there are digraphs with articulation points, but no bridges.

See also IsBiconnectedDigraph (6.3.4) and IsBridgelessDigraph (6.3.5).
Example

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

5.3.15 Bridges

> Bridges(D)

\textbf{Returns:} A (possibly empty) list of edges.

A connected digraph is \textit{2-edge-connected} if it is still connected (in the sense of \textit{IsConnectedDigraph} (6.3.3)) when any edge is removed. If the digraph $D$ is not 2-edge-connected but is connected, then any edge $[u, v]$ of $D$ whose removal makes the resulting digraph disconnected is called a \textit{bridge}.

Bridges returns a list of the bridges of $D$, if any, and, in particular, returns the empty list if $D$ is not connected.

Multiple edges are ignored by this method.

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

If $D$ has a bridge, then a node incident to the bridge is an articulation point (see \textit{ArticulationPoints} (5.3.14)) if and only if it has degree at least 2. It follows that if $D$ has a bridge and at least 3 nodes, then at least one of the nodes incident to the bridge is an articulation point. The converse does not hold, there are digraphs with articulation points, but no bridges.

See also \textit{IsBiconnectedDigraph} (6.3.4) and \textit{IsBridgelessDigraph} (6.3.5).

Example

\begin{verbatim}
gap> D := Digraph([[2, 5], [1, 3, 4, 5], [2, 4], [2, 3], [1, 2]]);
<immutable digraph with 5 vertices, 12 edges>
gap> Bridges(D);
[ ]
gap> D := Digraph([[2], [3], [4], [2]]);
<immutable digraph with 4 vertices, 4 edges>
gap> Bridges(D);
[ [ 1, 2 ] ]
gap> Bridges(ChainDigraph(2));
[ [ 1, 2 ] ]
gap> ArticulationPoints(ChainDigraph(2));
[ ]
\end{verbatim}
5.3.16 StrongOrientation

> StrongOrientation(D) (operation)
> StrongOrientationAttr(D) (attribute)

Returns: A digraph or fail.

A strong orientation of a connected symmetric digraph $D$ (if it exists) is a strongly connected subdigraph $C$ of $D$ such that for every edge $[u, v]$ of $D$ either $[u, v]$ or $[v, u]$ is an edge of $C$ but not both. Robbin’s Theorem states that a digraph admits a strong orientation if and only if it is bridgeless (see IsBridgelessDigraph (6.3.5)).

This operation returns a strong orientation of the digraph $D$ if $D$ is symmetric and $D$ admits a strong orientation. If $D$ is symmetric but does not admit a strong orientation, then fail is returned. If $D$ is not symmetric, then an error is given.

If $D$ is immutable, StrongOrientation($D$) returns an immutable digraph, and if $D$ is mutable, then StrongOrientation($D$) returns a mutable digraph.

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

Example

```gap
> StrongOrientation(DigraphSymmetricClosure(CycleDigraph(5))); = CycleDigraph(5);
true
> D := DigraphSymmetricClosure(Digraph([[[2, 7], [3, 5], [4], [2], [6], [1], []]]));
> IsBridgelessDigraph(D);
false
> StrongOrientation(D);
fail
> StrongOrientation(NullDigraph(0));
<immutable empty digraph with 0 vertices>
> StrongOrientation(DigraphDisjointUnion(CompleteDigraph(3),
> CompleteDigraph(3)));
fail
```

5.3.17 DigraphPeriod

> DigraphPeriod(digraph) (attribute)

Returns: An integer.

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

If a digraph $digraph$ has at least one directed cycle, then the period is the greatest positive integer which divides the lengths of all directed cycles of $digraph$. If $digraph$ 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 IsAperiodicDigraph (6.3.7).

Example

```gap
> D := Digraph([[6], [1], [2], [3], [4, 4], [5]]);
<immutable multidigraph with 6 vertices, 7 edges>
> DigraphPeriod(D);
6
> D := Digraph([[2], [3, 5], [4], [5], [1, 2]]);
<immutable digraph with 5 vertices, 7 edges>
> DigraphPeriod(D);
1
```
Digraphs

5.3.18 DigraphFloydWarshall

\texttt{DigraphFloydWarshall(digraph, func, nopath, edge)}

\textbf{Returns:} A matrix.

If \texttt{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 \texttt{digraph}.

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

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

\begin{verbatim}
  for i in DigraphsVertices(digraph) do
    for j in DigraphsVertices(digraph) do
      for k in DigraphsVertices(digraph) do
        func(mat, i, j, k);
        od;
      od;
    od;
\end{verbatim}

The matrix \texttt{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:

\begin{verbatim}
Example

\texttt{gap> D := DigraphFromDigraph6String("\&EAHQeDB");}
<immutable digraph with 6 vertices, 12 edges>
\texttt{gap> func := function(mat, i, j, k)
     > if mat[i][k] <> -1 and mat[k][j] <> -1 then
     >   if (mat[i][j] = -1) or (mat[i][j] > mat[i][k] + mat[k][j]) then
     >     mat[i][j] := mat[i][k] + mat[k][j];
     >   fi;
     >   fi;
     > end;
   function(mat, i, j, k)
\texttt{gap> shortest_distances := DigraphFloydWarshall(D, func, -1, 1);};
\texttt{gap> Display(shortest_distances);
[ [ 3, -1, -1, 2, 1, 2 ],}
\end{verbatim}
5.3.19 IsReachable

\$\textbf{IsReachable(digraph, u, v)}$ (operation)

**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 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);
fake
gap> IsReachable(D, 3, 3);
true
gap> IsReachable(D, 1, 1);
fake
```

5.3.20 DigraphPath

\$\textbf{DigraphPath(digraph, u, v)}$ (operation)

**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 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 $n-1$, $\langle v_1, e_1, v_2, e_2, ..., e_{n-1}, v_n \rangle$, is represented by a pair of lists $[v, a]$ as follows:

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

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

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

5.3.21 DigraphShortestPath

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

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

Returns the shortest directed path in the digraph 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 fail. See Section ‘Definitions’ for the definition of a directed path and a directed cycle.

See DigraphPath (5.3.20) for details on the output. The method for DigraphShortestPath has worst case complexity of \( O(m + n) \) where \( m \) is the number of edges and \( n \) the number of vertices in digraph.

Example

\begin{verbatim}
gap> D := Digraph([[1, 2, 4, 1], [3, 5], [2, 3], [1, 2], [4]]);
<immutable digraph with 5 vertices, 11 edges>
gap> DigraphShortestPath(D, 1, 4);
[ [ 1, 4 ], [ 2 ] ]
\end{verbatim}

5.3.22 IteratorOfPaths

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

\textbf{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 (5.3.20) 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{verbatim}
gap> D := Digraph([[1, 4, 2], [3, 5], [2, 3], [1, 2], [4]]);
<immutable multidigraph with 5 vertices, 11 edges>
gap> iter := IteratorOfPaths(D, 1, 4);
<iterator>
gap> NextIterator(iter);
[ [ 1, 4 ], [ 2 ] ]
\end{verbatim}
5.3.23 DigraphAllSimpleCircuits

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

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

If \textit{digraph} is a digraph, then \textbf{DigraphAllSimpleCircuits} returns a list of the \textit{simple circuits} in \textit{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 \textbf{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, \textbf{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 $(v_1,e_1,v_2,e_2,...,e_{n-1},v_n,e_{n+1},v_1)$ can be represented as the list $[v_1,...,v_n]$, or any cyclic permutation thereof. For each simple circuit of \textit{digraph}, \textbf{DigraphAllSimpleCircuits}(\textit{digraph}) includes precisely one such list to represent the circuit.

\begin{verbatim}
Example

gap> D := Digraph([[1, 3], [2, 4], [5, 4], [4]]);
<immutable digraph with 5 vertices, 6 edges>
digraphAllSimpleCircuits(D);
[ [ 4 ], [ 4, 5 ], [ 2, 3 ] ]

gap> D := ChainDigraph(10);
<immutable digraph with 3 vertices, 3 edges>
digraphAllSimpleCircuits(D);
[ [ ] ]

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

gap> D := Digraph([[1, 1]]);
<immutable multidigraph with 1 vertex, 2 edges>
digraphAllSimpleCircuits(D);
[ [ 1 ] ]

gap> D := CycleDigraph(IsMutableDigraph, 3);
<mutable digraph with 3 vertices, 3 edges>
digraphAllSimpleCircuits(D);
[ [ 1, 2, 3 ] ]
\end{verbatim}
5.3.24 DigraphLongestSimpleCircuit

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

**Returns:** A list of vertices, or fail.

If \text{digraph} is a digraph, then \text{DigraphLongestSimpleCircuit} returns the longest \textit{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}(\text{digraph}) 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} (5.3.23).

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.

\begin{verbatim}
gap> D := Digraph([[], [3], [2, 4], [5, 4], [4]]);;
gap> DigraphLongestSimpleCircuit(D);
[ 4, 5 ]
gap> D := ChainDigraph(10);;
gap> DigraphLongestSimpleCircuit(D);
fail
gap> D := Digraph([[3], [1], [1, 4], [1, 1]]);;
gap> DigraphLongestSimpleCircuit(D);
[ 1, 3, 4 ]
gap> D := GeneralisedPetersenGraph(IsMutableDigraph, 4, 1);
<mutable digraph with 8 vertices, 24 edges>
gap> DigraphLongestSimpleCircuit(D);
[ 1, 2, 3, 4, 8, 7, 6, 5 ]
\end{verbatim}

5.3.25 DigraphLayers

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

**Returns:** A list.

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.

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

5.3.26 DigraphDegeneracy

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

**Returns:** A non-negative integer, or 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 fail.
Example

```gap
gap> D := DigraphSymmetricClosure(ChainDigraph(5));;
1
gap> D := CompleteDigraph(5);

gap> DigraphDegeneracy(D);
4

gap> D := Digraph([[1], [2, 4, 5], [3, 4], [2, 3, 4], [2], []]);
<immutable digraph with 6 vertices, 10 edges>

gap> DigraphDegeneracy(D);
1

gap> D := GeneralisedPetersenGraph(IsMutableDigraph, 10, 3);
<mutable digraph with 20 vertices, 60 edges>

gap> DigraphDegeneracy(D);
3
```

### 5.3.27 DigraphDegeneracyOrdering

>` DigraphDegeneracyOrdering(digraph) (attribute)

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

If *digraph* is a digraph for which DigraphDegeneracy(*digraph*) is a non-negative integer *k* - see DigraphDegeneracy (5.3.26) - then this attribute returns a degeneracy ordering of the vertices of the digraph.

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

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

Example

```gap
gap> D := DigraphSymmetricClosure(ChainDigraph(5));;

gap> 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, 10, 3);
<mutable digraph with 20 vertices, 60 edges>

gap> DigraphDegeneracyOrdering(D);
[ 6, 5, 4, 1, 3, 2 ]
```

### 5.3.28 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 (5.3.23). 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
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);  
fail
gap> D := GeneralisedPetersenGraph(IsMutableDigraph, 5, 2);  
<mutable digraph with 10 vertices, 30 edges>
gap> HamiltonianPath(D);  
fail
```

5.3.29 NrSpanningTrees

- NrSpanningTrees(digraph)

**Returns:** An integer.

Returns the number of spanning trees of the symmetric digraph digraph. NrSpanningTrees will return an error if digraph is not a symmetric digraph.

See IsSymmetricDigraph (6.1.12) and IsUndirectedSpanningTree (4.1.2) for more information.

Example

```gap
gap> D := CompleteDigraph(5);  
<immutable complete digraph with 5 vertices>
gap> NrSpanningTrees(D);  
125
gap> D := DigraphSymmetricClosure(CycleDigraph(24));  
gap> NrSpanningTrees(D);  
24
gap> NrSpanningTrees(EmptyDigraph(0));  
0
gap> D := GeneralisedPetersenGraph(IsMutableDigraph, 9, 2);  
<mutable digraph with 18 vertices, 54 edges>
gap> NrSpanningTrees(D);  
1134225
```
5.3.30 DigraphDijkstra (for a source and target)

- DigraphDijkstra(digraph, source, target) (operation)
- DigraphDijkstra(digraph, source) (operation)

Returns: Two lists.

If digraph is a digraph and source and target are vertices of digraph, then DigraphDijkstra calculates the length of the shortest path from source to target and returns two lists. Each element of the first list is the distance of the corresponding element from source. If a vertex was not visited in the process of calculating the shortest distance to target or if there is no path connecting that vertex with source, then the corresponding distance is infinity. Each element of the second list gives the previous vertex in the shortest path from source to the corresponding vertex. For source and for any vertices that remained unvisited this will be -1.

If the optional second argument target is not present, then DigraphDijkstra returns the shortest path from source to every vertex that is reachable from source.

Example

```gap
gap> mat := [[0, 1, 1], [0, 0, 1], [0, 0, 0]];  # adjacency matrix
[ [ 0, 1, 1 ], [ 0, 0, 1 ], [ 0, 0, 0 ] ]
gap> D := DigraphByAdjacencyMatrix(mat);  # create digraph
<immutable digraph with 3 vertices, 3 edges>
gap> DigraphDijkstra(D, 2, 3);  # shortest path from 2 to 3
[ [ infinity, 0, 1 ], [ -1, -1, 2 ] ]
gap> DigraphDijkstra(D, 1, 3);  # shortest path from 1 to 3
[ [ 0, 1, 1 ], [ -1, 1, 1 ] ]
gap> DigraphDijkstra(D, 1, 2);  # shortest path from 1 to 2
[ [ 0, 1, 1 ], [ -1, 1, 1 ] ]
```

5.4 Cayley graphs of groups

5.4.1 GroupOfCayleyDigraph

- GroupOfCayleyDigraph(digraph) (attribute)
- SemigroupOfCayleyDigraph(digraph) (attribute)

Returns: A group or semigroup.

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

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

See also GeneratorsOfCayleyDigraph (5.4.2).

Example

```gap
gap> G := DihedralGroup(IsPermGroup, 8);  # dihedral group
Group([ (1,2,3,4), (2,4) ])
gap> digraph := CayleyDigraph(G);  # Cayley graph
<immutable digraph with 8 vertices, 16 edges>
gap> GroupOfCayleyDigraph(digraph) = G;  # check
true
```
5.4.2 GeneratorsOfCayleyDigraph

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

**Returns:** A list of generators.

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

See also \(\text{GroupOfCayleyDigraph}\) (5.4.1) or \(\text{SemigroupOfCayleyDigraph}\) (5.4.1).

\[
\text{Example}\n\]

\[
gap> G := DihedralGroup(IsPermGroup, 8);\n\text{Group([ (1,2,3,4), (2,4) ])}\n\gap> digraph := CayleyDigraph(G);\n<\text{immutable digraph with 8 vertices, 16 edges}>\n\gap> GeneratorsOfCayleyDigraph(digraph) = GeneratorsOfGroup(G);\ntrue\n\gap> digraph := CayleyDigraph(G, [()]);\n<\text{immutable digraph with 8 vertices, 8 edges}>\n\gap> GeneratorsOfCayleyDigraph(digraph) = [()];\ntrue\n\]

5.5 Associated semigroups

5.5.1 AsSemigroup (for a filter and a digraph)

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

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

**Returns:** A semilattice of partial perms.

The operation \(\text{AsSemigroup}\) requires that \(\text{filt}\) be equal to \(\text{IsPartialPermSemigroup}\) (Reference: \(\text{IsPartialPermSemigroup}\)). If \(\text{digraph}\) is a \(\text{IsJoinSemilatticeDigraph}\) (6.1.17) or \(\text{IsLatticeDigraph}\) (6.1.17) then \(\text{AsSemigroup}\) returns a semigroup of partial perms which is isomorphic to the semigroup whose elements are the vertices of \(\text{digraph}\) with the binary operation \(\text{PartialOrderDigraphJoinOfVertices}\) (5.2.15). If \(\text{digraph}\) satisfies \(\text{IsMeetSemilatticeDigraph}\) (6.1.17) but not \(\text{IsJoinSemilatticeDigraph}\) (6.1.17) then \(\text{AsSemigroup}\) returns a semigroup of partial perms which is isomorphic to the semigroup whose elements are the vertices of \(\text{digraph}\) with the binary operation \(\text{PartialOrderDigraphMeetOfVertices}\) (5.2.15).

The operation \(\text{AsMonoid}\) behaves similarly to \(\text{AsSemigroup}\) except that \(\text{filt}\) may also be equal to \(\text{IsPartialPermMonoid}\) (Reference: \(\text{IsPartialPermMonoid}\)), \(\text{digraph}\) must satisfy \(\text{IsLatticeDigraph}\) (6.1.17), and the output satisfies \(\text{IsMonoid}\) (Reference: \(\text{IsMonoid}\)).

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

\[
\text{Example}\n\]

\[
gap> di := Digraph([[1], [1, 2], [1, 3], [1, 4], [1, 2, 3, 5]]);\n<\text{immutable digraph with 5 vertices, 11 edges}>\n\gap> S := AsSemigroup(IsPartialPermSemigroup, di);\n<\text{partial perm semigroup of rank 3 with 4 generators}>\n\gap> ForAll(Elements(S), IsIdempotent);\ntrue\n\]
true
gap> IsInverseSemigroup(S);
true
gap> Size(S);
5
gap> di := Digraph([[1], [1, 2], [1, 2, 3]]);
<immutable digraph with 3 vertices, 6 edges>
gap> M := AsMonoid(IsPartialPermMonoid, di);
<partial perm monoid of rank 2 with 3 generators>
gap> Size(M);
3

5.5.2 AsSemigroup (for a filter, semilattice digraph, and two lists)

> AsSemigroup(filt, Y, gps, homs)

(operation)
Returns: A Clifford semigroup of partial perms.

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

Example

gap> G1 := AlternatingGroup(4);
gap> G2 := SymmetricGroup(2);
gap> G3 := SymmetricGroup(3);
gap> gr := Digraph([[1, 3], [2, 3], [3]]);
gap> sgn := function(x)
> if SignPerm(x) = 1 then
> return (1, 2);
> fi;
> return (1, 2);  
> end;;
gap> hom13 := GroupHomomorphismByFunction(G1, G3, sgn);;
gap> hom23 := GroupHomomorphismByFunction(G2, G3, sgn);;
gap> T := AsSemigroup(IsPartialPermSemigroup,
> gr, 
> [G1, G2, G3], [[1, 3, hom13], [2, 3, hom23]]);
gap> Size(T);
20
gap> D := GreensDClasses(T);
gap> List(D, x -> Size(x));
[ 6, 12, 2 ]
5.6 Planarity

5.6.1 KuratowskiPlanarSubdigraph

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

**Returns:** A list or fail.

KuratowskiPlanarSubdigraph returns the immutable list of lists of out-neighbours of a (not necessarily induced) subdigraph of the digraph `digraph` that witnesses the fact that `digraph` is not planar, or fail if `digraph` is planar. In other words, KuratowskiPlanarSubdigraph returns the out-neighbours of a subdigraph of `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> 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> KuratowskiPlanarSubdigraph(D);
fail
```

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

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

5.6.2 KuratowskiOuterPlanarSubdigraph

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

**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
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 ],
  [ 6 ], [ 11 ], [ ], [ ] ]
gap> D := Digraph([ [ 2, 4, 7, 9, 10 ], [ 3, 5, 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> KuratowskiOuterPlanarSubdigraph(D);
[ [ ], [ ], [ ], [ 8, 9 ], [ ], [ ], [ 9, 4 ], [ 7, 9 ], [ ],
  [ 6 ], [ 11 ], [ ], [ ] ]
gap> D := Digraph(IsMutableDigraph, [ [ 3, 5, 10 ], [ 8, 9, 10 ], [ 1, 4 ],
                   > [ 1, 7, 11 ], [ 4, 7 ], [ 6, 8 ], [ 2, 7 ], [ 2, 11 ], [ 1, 2 ], [ 5, 9 ] ]);;
<mutable digraph with 11 vertices, 25 edges>
gap> KuratowskiOuterPlanarSubdigraph(D);
[ [ 3, 5, 10 ], [ 9, 8, 10 ], [ 4 ], [ 6 ], [ 11 ], [ 7 ], [ 8 ],
  [ 6 ], [ 11 ], [ ], [ ] ]
gap> D := Digraph(IsMutableDigraph, [ [ 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 ] ]);;
<mutable digraph with 10 vertices, 50 edges>
gap> KuratowskiOuterPlanarSubdigraph(D);
false
```

5.6.3 PlanarEmbedding

> PlanarEmbedding(digraph) (attribute)

Returns: A list or fail.

If `digraph` is a planar digraph, then PlanarEmbedding 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 \texttt{digraph} is not planar, then \texttt{fail} is returned.

The directions and multiplicities of any edges in \texttt{digraph} are ignored by \texttt{PlanarEmbedding}.
See also \texttt{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

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

5.6.4 \texttt{OuterPlanarEmbedding}

\textbf{OuterPlanarEmbedding(digraph)}

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

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

The directions and multiplicities of any edges in \texttt{digraph} are ignored by \texttt{OuterPlanarEmbedding}.
See also \texttt{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

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


> [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> 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>
> gap> OuterPlanarEmbedding(D);
> fail
> gap> D := Digraph(IsMutableDigraph, [[2, 4, 7, 9, 10],
> [1, 3, 4, 6, 9, 10], [6, 10], [2, 5, 7, 9, 10],
> [3, 4, 5, 6, 9, 10], [3, 4, 5, 7, 9], [2, 3, 5, 6, 7, 8], [3, 5]]);
> <mutable digraph with 10 vertices, 50 edges>
> gap> OuterPlanarEmbedding(D);
> fail
> gap> OuterPlanarEmbedding(CompleteBipartiteDigraph(2, 2));
> [ [ 3, 4 ], [ 4, 3 ], [ ], [ ] ]

5.6.5 SubdigraphHomeomorphicToK23

SubdigraphHomeomorphicToK23(digraph)
SubdigraphHomeomorphicToK23(digraph)
SubdigraphHomeomorphicToK4(digraph)

Returns: A list or fail.

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].

Example

> 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> 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([[3, 5, 10], [8, 9, 10], [1, 4], [3, 6], [1, 11],
> [4, 7], [6, 8], [2, 7], [2, 11], [1, 2], [5, 9]]);
> <immutable digraph with 11 vertices, 24 edges>
> gap> SubdigraphHomeomorphicToK4(D);
> fail
> gap> SubdigraphHomeomorphicToK23(D);
Digraphs

[ [ 3, 10, 5 ], [ 10, 8, 9 ], [ 4 ], [ 6 ], [ 11 ], [ 7 ], [ 8 ], 
  [ ], [ 11 ], [ ], [ ] ]

gap> SubdigraphHomeomorphicToK33(D);
fail

gap> SubdigraphHomeomorphicToK23(NullDigraph(0));
fail

gap> SubdigraphHomeomorphicToK33(CompleteDigraph(5));
fail

gap> SubdigraphHomeomorphicToK33(CompleteBipartiteDigraph(3, 3));

[ [ 4, 6, 5 ], [ 4, 5, 6 ], [ 6, 5, 4 ], [ ], [ ], [ ] ]

gap> SubdigraphHomeomorphicToK4(CompleteDigraph(3));
fail

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);

[ [ 3, 10, 5 ], [ 10, 8, 9 ], [ 4 ], [ 6 ], [ 11 ], [ 7 ], [ 8 ], 
  [ ], [ 11 ], [ ], [ ] ]

gap> SubdigraphHomeomorphicToK33(D);
fail

gap> SubdigraphHomeomorphicToK23(NullDigraph(0));
fail

gap> SubdigraphHomeomorphicToK33(CompleteDigraph(5));
fail

gap> SubdigraphHomeomorphicToK33(CompleteBipartiteDigraph(3, 3));

[ [ 4, 6, 5 ], [ 4, 5, 6 ], [ 6, 5, 4 ], [ ], [ ], [ ] ]

gap> SubdigraphHomeomorphicToK4(CompleteDigraph(3));
fail
Chapter 6

Properties of digraphs

6.1 Edge properties

6.1.1 DigraphHasLoops

\( \textbf{DigraphHasLoops}(\text{digraph}) \) (property)

\textbf{Returns:} true or false.

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

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> 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
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
gap> D := CompleteDigraph(IsMutableDigraph, 4);
<mutable digraph with 4 vertices, 12 edges>
gap> DigraphHasLoops(D);
false
\end{verbatim}

6.1.2 IsAntiSymmetricDigraph

\( \textbf{IsAntiSymmetricDigraph}(\text{digraph}) \) (property)

\( \textbf{IsAntiSymmetricDigraph}(\text{digraph}) \) (property)

\textbf{Returns:} true or false.

This property is true if the digraph \textit{digraph} is antisymmetric, and false if it is not.

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

\begin{verbatim}
Example

\texttt{gap> gr1 := Digraph([[2], [1, 3], [2, 3]]);}
<immutable digraph with 3 vertices, 5 edges>
\texttt{gap> IsAntisymmetricDigraph(gr1);}
false
\texttt{gap> DigraphEdges(gr1){[1, 2]};}
[ [1, 2], [2, 1] ]
\texttt{gap> gr2 := Digraph([[1, 2], [3, 3], [1]]);}
<immutable multidigraph with 3 vertices, 5 edges>
\texttt{gap> IsAntisymmetricDigraph(gr2);}
true
\texttt{gap> DigraphEdges(gr2);}
[ [1, 1], [1, 2], [2, 3], [2, 3], [3, 1] ]
\end{verbatim}

\subsection{6.1.3 \texttt{IsBipartiteDigraph}}

\begin{verbatim}
Example

\texttt{gap> D := ChainDigraph(4);}
<immutable chain digraph with 4 vertices>
\texttt{gap> IsBipartiteDigraph(D);}
true
\texttt{gap> D := CycleDigraph(3);}
<immutable cycle digraph with 3 vertices>
\texttt{gap> IsBipartiteDigraph(D);}
true
\texttt{gap> D := CompleteBipartiteDigraph(IsMutableDigraph, 5, 4);}
<mutable digraph with 9 vertices, 40 edges>
\texttt{gap> IsBipartiteDigraph(D);}
true
\end{verbatim}

\subsection{6.1.4 \texttt{IsCompleteBipartiteDigraph}}

\begin{verbatim}
Example

\texttt{gap> D := ChainDigraph(4);}
<immutable chain digraph with 4 vertices>
\texttt{gap> IsBipartiteDigraph(D);}
true
\texttt{gap> D := CycleDigraph(3);}
<immutable cycle digraph with 3 vertices>
\texttt{gap> IsBipartiteDigraph(D);}
true
\texttt{gap> D := CompleteBipartiteDigraph(IsMutableDigraph, 5, 4);}
<mutable digraph with 9 vertices, 40 edges>
\texttt{gap> IsBipartiteDigraph(D);}
true
\end{verbatim}
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 \( \text{digraph} \) is mutable, then the return value of this property is recomputed every time it is called.

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

6.1.5 \texttt{IsCompleteDigraph}

\textbf{Example}

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

6.1.6 \texttt{IsCompleteMultipartiteDigraph}

\textbf{Returns}: true or false.

This property returns true if \( \text{digraph} \) 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.

Example

```
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

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

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

Returns: true or false.

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

IsNullDigraph is a synonym for IsEmptyDigraph.

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

Example

```
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

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

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.
6.1.9 IsFunctionalDigraph

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

Returns: true or false.

This property is true if the digraph \textit{digraph} is functional.

A digraph is \textit{functional} if every vertex is the source of a unique edge.

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> 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
\end{verbatim}

6.1.10 IsMultiDigraph

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

Returns: true or false.

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

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> 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

gap> IsMultiDigraph(D);
\end{verbatim}
6.1.11 IsReflexiveDigraph

\textbf{IsReflexiveDigraph} \texttt{(digraph)}

\textbf{Returns:} true or false.

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

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, 2], [2]]);  # immutable digraph with 2 vertices, 3 edges
<immutable digraph with 2 vertices, 3 edges>
gap> IsReflexiveDigraph(D);       # true
true

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

6.1.12 IsSymmetricDigraph

\textbf{IsSymmetricDigraph} \texttt{(digraph)}

\textbf{Returns:} true or false.

This property is true if the digraph \texttt{digraph} is symmetric, and false if it is not. A symmetric digraph is one where for each non-loop edge, having source \texttt{u} and range \texttt{v}, there is a corresponding edge with source \texttt{v} and range \texttt{u}. If there are \(n\) edges with source \texttt{u} and range \texttt{v}, then there must be precisely \(n\) edges with source \texttt{v} and range \texttt{u}. In other words, a symmetric digraph has a symmetric adjacency matrix \texttt{AdjacencyMatrix} (5.2.1).

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

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

gap> adj1 := AdjacencyMatrix(gr1);  # boolean
true

gap> Display(adj1);
[ [ 0, 1, 0 ],
  [ 1, 0, 1 ],
  [ 0, 1, 1 ] ]
gap> adj1 = TransposedMat(adj1);   # true
true

gap> gr1 := DigraphReverse(gr1);  # symmetric
true

gap> gr2 := Digraph([[2, 3], [1, 3], [2, 3]]);
<mutable digraph with 3 vertices, 5 edges>
\end{verbatim}
6.1.13 IsTournament

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

\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

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

\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
Digraphs 105

DigraphTopologicalSort (5.1.7), then methods with complexity $O(m + n + m \cdot n)$ will be used when appropriate.

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

Example

```gap
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
```

6.1.15 IsPreorderDigraph

▷ IsPreorderDigraph(digraph) (property)
▷ IsQuasiorderDigraph(digraph) (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) digraph corresponds to the preorder relation $\leq$ defined by $x \leq y$ if and only if $[x, y]$ is an edge of 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 := Digraph([[1], [2, 3], [2, 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);
false
gap> D := Digraph([[1], [1, 2], [2, 3]]);
<immutable digraph with 3 vertices, 5 edges>
gap> IsQuasiorderDigraph(D);
false
```
6.1.16  IsPartialOrderDigraph

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

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

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

```gap
gap> D := Digraph([[1, 3], [2, 3], [3]]);
<immutable digraph with 3 vertices, 5 edges>
gap> IsPartialOrderDigraph(D);
true
gap> D := CycleDigraph(5);
<immutable cycle digraph with 5 vertices>
gap> IsPartialOrderDigraph(D);
false
```  

6.1.17  IsMeetSemilatticeDigraph

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

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

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

**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 meet semilattice if it is a partial order and every pair of vertices has a greatest lower bound (meet) with respect to the aforementioned relation. A join semilattice is a partial order where every pair of vertices has a least upper bound (join) with respect to the relation.

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

```gap
gap> D := Digraph([[1, 3], [2, 3], [3]]);
<immutable digraph with 3 vertices, 5 edges>
gap> IsMeetSemilatticeDigraph(D);
false
gap> IsJoinSemilatticeDigraph(D);
true
gap> IsLatticeDigraph(D);
false
```
6.2 Regularity

6.2.1 IsInRegularDigraph

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

\textbf{Returns:} true or false.

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

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

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

\begin{verbatim}
gap> IsInRegularDigraph(ChainDigraph(4));
false
\end{verbatim}

6.2.2 IsOutRegularDigraph

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

\textbf{Returns:} true or false.

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

See also IsInRegularDigraph (6.2.1) 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.

\begin{verbatim}
gap> IsOutRegularDigraph(CompleteDigraph(4));
true
gap> IsOutRegularDigraph(ChainDigraph(4));
false
\end{verbatim}

### 6.2.3 \texttt{IsRegularDigraph}

\begin{verbatim}
\texttt{IsRegularDigraph}(\texttt{digraph})
\end{verbatim}

\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 and terminating at \(v\). In other words, the property is true if \textit{digraph} is both in-regular and out-regular. See also \texttt{IsInRegularDigraph} (6.2.1) and \texttt{IsOutRegularDigraph} (6.2.2).

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

\begin{verbatim}
gap> IsRegularDigraph(CompleteDigraph(4));
true
gap> IsRegularDigraph(ChainDigraph(4));
false
\end{verbatim}

### 6.2.4 \texttt{IsDistanceRegularDigraph}

\begin{verbatim}
\texttt{IsDistanceRegularDigraph}(\texttt{digraph})
\end{verbatim}

\textbf{Returns:} true or false.

If \textit{digraph} is a connected symmetric graph, this property returns true if for any two vertices \(u\) and \(v\) of digraph \textit{digraph} and any two integers \(i\) and \(j\) between 0 and the diameter of \textit{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 \textit{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 \textit{digraph} is distance regular.

In the case where \textit{digraph} is not symmetric or not connected, the property is false.

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 := DigraphSymmetricClosure(ChainDigraph(5));;
gap> IsDistanceRegularDigraph(D);
false
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
\end{verbatim}
6.3 Connectivity and cycles

6.3.1 IsAcyclicDigraph

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

**Returns:** true or false.

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

```
Example

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

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

gap> D := ChainDigraph(10);
<mutable digraph with 10 vertices, 9 edges>
gap> IsChainDigraph(D);
true

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

gap> D := CycleDigraph(10);
<immutable digraph with 10 vertices, 10 edges>
```

6.3.2 IsChainDigraph

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

**Returns:** true or false.

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

A digraph is a chain if and only if it is a directed tree, in which every vertex has out degree at most one; see IsDirectedTree (6.3.8) and OutDegrees (5.2.8).

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

```
Example

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

Digraphs

gap> D := ChainDigraph(5);
<immutable chain digraph with 5 vertices>
gap> IsChainDigraph(D);
true
gap> D := DigraphReverse(D);
<immutable digraph with 5 vertices, 4 edges>
gap> IsChainDigraph(D);
true
gap> D := ChainDigraph(IsMutableDigraph, 5);
<mutable digraph with 5 vertices, 4 edges>
gap> IsChainDigraph(D);
true
gap> D := DigraphReverse(D);
<mutable digraph with 5 vertices, 4 edges>
gap> IsChainDigraph(D);
true

6.3.3 IsConnectedDigraph

> IsConnectedDigraph(digraph) (property)

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

The method used in this function has complexity $O(m)$ if the digraph’s DigraphSource (5.2.5) 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 digraph is mutable, then the return value of this property is recomputed every time it is called.

Example

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

6.3.4 IsBiconnectedDigraph

> IsBiconnectedDigraph(digraph) (property)

Returns: true or false.
A connected digraph is biconnected if it is still connected (in the sense of IsConnectedDigraph (6.3.3)) when any vertex is removed. If $D$ has at least 3 vertices, then IsBiconnectedDigraph
implies IsBridgelessDigraph (6.3.5); see ArticulationPoints (5.3.14) or Bridges (5.3.15) for a more detailed explanation.

IsBiconnectedDigraph returns true if the digraph digraph is biconnected, and false if it is not. In particular, IsBiconnectedDigraph returns false if digraph is not connected.

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

Multiple edges are ignored by this method.

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

See also Bridges (5.3.15), ArticulationPoints (5.3.14), and IsBridgelessDigraph (6.3.5).

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]]);

Example

```
6.3.6 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> IsStronglyConnectedDigraph(IsMutableDigraph, 250000);
true
gap> DigraphRemoveEdge(D, [250000, 1]);
<mutable digraph with 250000 vertices, 249999 edges>
gap> IsStronglyConnectedDigraph(D);
false
```
6.3.7 IsAperiodicDigraph

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

\textbf{Returns:} true or false.

This property is true if the digraph \textit{digraph} is aperiodic, i.e. if its \textit{DigraphPeriod} (5.3.17) is equal to 1. Otherwise, the property is false.

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> 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
\end{verbatim}

6.3.8 IsDirectedTree

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

\textbf{Returns:} true or false.

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

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> D := Digraph([[6], [1], [2]]);
<immutable digraph with 2 vertices, 1 edge>
gap> IsDirectedTree(D);
false
gap> D := Digraph([[3], [3], [1]]);
<immutable digraph with 3 vertices, 2 edges>
gap> IsDirectedTree(D);
false
gap> D := Digraph([[2], [3], [1]]);
<immutable digraph with 3 vertices, 2 edges>
gap> IsDirectedTree(D);
true
\end{verbatim}
6.3.9 IsUndirectedTree

\[ \text{IsUndirectedTree}(\text{digraph}) \]
\[ \text{IsUndirectedForest}(\text{digraph}) \]

**Returns:** true or false.

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

An **undirected tree** is a symmetric digraph without loops, in which for any pair of distinct vertices \( u \) and \( v \), there is exactly one directed path from \( u \) to \( v \). See IsSymmetricDigraph (6.1.12) 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
```

6.3.10 IsEulerianDigraph

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

**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}

6.3.11 \textbf{IsHamiltonianDigraph}

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

\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}

\textbf{Returns:} \texttt{true} or \texttt{false}.

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

A digraph with \textit{n} vertices is \textit{Hamiltonian} if it has a directed cycle of length \textit{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}
\textbf{Example}
\end{verbatim}
6.3.12 IsCycleDigraph

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

**Returns:** true or false.

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

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

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

**Example**

```gap
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
```

6.4 Planarity

6.4.1 IsPlanarDigraph

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

**Returns:** true or false.

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

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

See also `IsOuterPlanarDigraph` (6.4.2).

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

**Example**

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

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

\texttt{IsOuterPlanarDigraph} returns \texttt{true} if the digraph \texttt{digraph} is outer planar and \texttt{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 edge-addition-planarity-suite by John Boyer of the algorithms described in [BM06].
6.5 Homomorphisms and transformations

6.5.1 IsDigraphCore

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

*property*

**Returns:** true or false.

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

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

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

```
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
```

6.5.2 IsEdgeTransitive

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

*property*

**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.

```
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,
```

6.5.3 IsVertexTransitive

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

*property*

**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
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)

⊿ OnDigraphs(digraph, perm)  (operation)
⊿ OnDigraphs(digraph, trans)  (operation)

Returns: A digraph.

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

If the second argument is a transformation trans of the vertices of digraph, then this operation returns a digraph constructed by transforming the source and range of each edge according to trans. Thus a vertex which does not appear in the image of trans will be isolated in the returned digraph, and the returned digraph may contain multiple edges, even if digraph does not.

If trans is mathematically a permutation, then the result coincides with OnDigraphs(digraph, AsPermutation(trans)).

The DigraphVertexLabels (5.1.9) of digraph will not be retained in the returned digraph.

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

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

\begin{itemize}
\item OnMultiDigraphs(digraph, pair) (operation)
\item OnMultiDigraphs(digraph, perm1, perm2) (operation)
\end{itemize}

**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 `perm1` and `perm2`, 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.

```
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 labellings

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

7.2.1 DigraphsUseNauty

\begin{itemize}
\item DigraphsUseNauty() (function)
\item DigraphsUseBliss() (function)
\end{itemize}

**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.

```gap
gap> johnson := DigraphFromGraph6String("E}lw");
gap> G := AutomorphismGroup(johnson);
Group([ (3,4), (2,3)(4,5), (1,2)(5,6) ])

gap> cycle := CycleDigraph(9);
<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.

```gap
gap> G := BlissAutomorphismGroup(JohnsonDigraph(5, 2));;
gap> IsSymmetricGroup(G);
true
gap> Size(G);
120
```

### 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.

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

### 7.2.5 AutomorphismGroup (for a digraph and a homogeneous list)

> 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 \textit{vert\_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 (Reference: 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)}.

\begin{verbatim}
gap> cycle := CycleDigraph(9);
<immutable cycle digraph with 9 vertices>
gap> G := AutomorphismGroup(cycle);

true

gap> colours := [[1, 4, 7], [2, 5, 8], [3, 6, 9]];

true

gap> H := AutomorphismGroup(cycle, colours);

true

Example

7.2.6 AutomorphismGroup (for a digraph, homogeneous list, and list)

\begin{verbatim}
gap> H = AutomorphismGroup(cycle, [1, 2, 3, 1, 2, 3, 1, 2, 3]);
true

gap> H = SubgroupByProperty(G, p -> OnTuplesSets(colours, p) = colours);
true

gap> IsTrivial(AutomorphismGroup(cycle, [1, 1, 2, 2, 2, 2, 2, 2, 2]));
true
\end{verbatim}

\end{verbatim}

\texttt{AutomorphismGroup(digraph, vert\_colours, edge\_colours)} (operation)

\textbf{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 \texttt{digraph} and colourings \texttt{vert\_colours}, \texttt{edge\_colours}. Let \( n \) be the number of vertices of \texttt{digraph}. The colourings must have the following forms:

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

- \texttt{edge\_colours} must be \texttt{fail} or a list of \( n \) lists of integers of the same shape as \texttt{OutNeighbours(digraph)}, where \texttt{edge\_colours}[i][j] is the colour of the edge \texttt{OutNeighbours(digraph)}[i][j], using the colours \([1 .. k]\) for some \( k \leq n \);

Giving \texttt{vert\_colours [edge\_colours]} as \texttt{fail} is equivalent to setting all vertices [edges] to be the same colour.

Unlike \texttt{AutomorphismGroup (7.2.2)}, it is possible to obtain the automorphism group of an edge-coloured multidigraph (see \texttt{IsMultiDigraph (6.1.10)}) when no two edges share the same source.
range, and colour. The automorphism group of a vertex/edge-coloured digraph \( digraph \) with colouring \( c \) is the group consisting of its vertex/edge-colour preserving automorphisms; an automorphism of \( digraph \) is an isomorphism of vertex/edge-coloured digraphs from \( digraph \) to itself. This group is equal to the subgroup of \( \text{AutomorphismGroup}(digraph) \) consisting of those automorphisms that preserve the colouring specified by \( colours \). See \( \text{AutomorphismGroup} \) (7.2.2), and see \( \text{IsomorphismDigraphs} \) (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 \( \text{Projection} \) (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 \( \text{BlissAutomorphismGroup} \) (7.2.3), \( \text{NautyAutomorphismGroup} \) (7.2.4), \( \text{DigraphsUseBliss} \) (7.2.1), and \( \text{DigraphsUseNauty} \) (7.2.1).

Example

\[
\text{gap> cycle := CycleDigraph(12);}
\text{<immutable cycle digraph with 12 vertices>}
\text{gap> vert_colours := List([1 .. 12], x -> x mod 3 + 1);;}
\text{gap> edge_colours := List([1 .. 12], x -> [x mod 2 + 1]);;}
\text{gap> Size(AutomorphismGroup(cycle));}
12
\text{gap> Size(AutomorphismGroup(cycle, vert_colours));}
4
\text{gap> Size(AutomorphismGroup(cycle, fail, edge_colours));}
6
\text{gap> Size(AutomorphismGroup(cycle, vert_colours, edge_colours));}
2
\text{gap> IsTrivial(AutomorphismGroup(cycle, > vert_colours, List([1 .. 12], x -> [x mod 4 + 1])));}
true
\]

7.2.7 BlissCanonicalLabelling (for a digraph)

▷ BlissCanonicalLabelling(digraph) (attribute)
▷ NautyCanonicalLabelling(digraph) (attribute)

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

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

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

A canonical labelling of a digraph \( G \) (under \( \rho \)) is an isomorphism of \( G \) onto its canonical representative, \( \rho(G) \). See \( \text{IsomorphismDigraphs} \) (7.2.17) for more information about isomorphisms of digraphs.

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

$\text{BlissCanonicalLabelling}$ can only be computed if $\text{digraph}$ has no multiple edges; see $\text{IsMultiDigraph}(6.1.10)$.

Example

```gap
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)) =
gap> OnDigraphs(digraph2, BlissCanonicalLabelling(digraph2));
true
```

### 7.2.8 BlissCanonicalLabelling (for a digraph and a list)

\[\text{BlissCanonicalLabelling}(\text{digraph}, \text{colours})\]  
\[\text{NautyCanonicalLabelling}(\text{digraph}, \text{colours})\]  

**Returns:** A permutation.

A function $\rho$ that maps a coloured digraph to a coloured digraph is a 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 canonical labelling of a coloured digraph $G$ (under $\rho$) is an isomorphism of $G$ onto its canonical representative, $\rho(G)$. See $\text{IsomorphismDigraphs}(7.2.18)$ for more information about isomorphisms of coloured digraphs.

A coloured digraph can be specified by its underlying digraph $\text{digraph}$ and its colouring $\text{colours}$. Let $n$ be the number of vertices of $\text{digraph}$. The colouring $\text{colours}$ may have one of the following two forms:

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

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

```gap
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:

```gap
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:

```gap
gap> OnTuplesSets(colours, p);
```

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

### 7.2.9 BlissCanonicalDigraph

- BlissCanonicalDigraph(`digraph`) (attribute)
- NautyCanonicalDigraph(`digraph`) (attribute)

**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 BlissCanonicalDigraph(`gr`) and BlissCanonicalDigraph(`digraph`) are equal if and only if `gr` and `digraph` are isomorphic as digraphs.
Analogously, the attribute `NautyCanonicalLabelling` returns the canonical representative found by applying `NautyCanonicalLabelling` (7.2.7).

If the argument `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>
gap> canon := BlissCanonicalDigraph(digraph);
<immutable digraph with 4 vertices, 7 edges>
gap> OutNeighbours(canon);
[ [ 1 ], [ 2 ], [ 3, 2 ], [ 1, 3, 2 ] ]
```

### 7.2.10 DigraphGroup

DigraphGroup(`digraph`)

**Returns:** A permutation group.

If `digraph` is immutable and was created knowing a subgroup of its automorphism group, then this group is stored in the attribute `DigraphGroup`. If `digraph` is mutable, or was not created knowing a subgroup of its automorphism group, then `DigraphGroup` returns the entire automorphism group of `digraph`. Note that if `digraph` is mutable, then the automorphism group is recomputed every time this function is called.

Note that certain other constructor operations such as `CayleyDigraph` (3.1.12), `BipartiteDoubleDigraph` (3.3.37), and `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 `DigraphGroup` attribute for the resulting digraph.

```
Example
gap> n := 4;;
gap> adj := function(x, y)
> return (((x - y) mod n) = 1) or (((x - y) mod n) = n - 1);
> end;
gap> group := CyclicGroup(IsPermGroup, n);
Group([ (1,2,3,4) ])
gap> D := Digraph(IsMutableDigraph, group, [1 .. n], ^, adj);
<mutable digraph with 4 vertices, 8 edges>
gap> HasDigraphGroup(D); false
gap> DigraphGroup(D);
Group([ (2,4), (1,2)(3,4) ])
gap> AutomorphismGroup(D);
Group([ (2,4), (1,2)(3,4) ])
gap> D := Digraph(group, [1 .. n], ^, adj);
<immutable digraph with 4 vertices, 8 edges>
gap> HasDigraphGroup(D); true
gap> DigraphGroup(D);
Group([ (1,2,3,4) ])
gap> D := DoubleDigraph(D);
<immutable digraph with 8 vertices, 32 edges>
gap> HasDigraphGroup(D); true
gap> DigraphGroup(D);
```
Digraphs

> Group( [(1,2,3,4)(5,6,7,8), (1,5)(2,6)(3,7)(4,8)] )
> AutomorphismGroup(D) =
> Group( [(6, 8), (5, 7), (4, 6), (3, 5), (2, 4),
> (1, 2)(3, 4)(5, 6)(7, 8)] );
true
> D := Digraph([[2, 3], [], []]);
<immutable digraph with 3 vertices, 2 edges>
> HasDigraphGroup(D);
false
> HasAutomorphismGroup(D);
false
> DigraphGroup(D);
Group([[2, 3]])
> HasAutomorphismGroup(D);
true
> group := DihedralGroup(8);
<pc group of size 8 with 3 generators>
> D := CayleyDigraph(group);
<immutable digraph with 8 vertices, 24 edges>
> HasDigraphGroup(D);
true
> GeneratorsOfGroup(DigraphGroup(D));
[ (1,2)(3,8)(4,6)(5,7), (1,3,4,7)(2,5,6,8), (1,4)(2,6)(3,7)(5,8) ]

7.2.11 DigraphOrbits

DigraphOrbits(digraph)

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 digraph.

Example

> G := Group( [(2, 3)(7, 8, 9), (1, 2, 3)(4, 5, 6)(8, 9)] );
> D := EdgeOrbitsDigraph(G, [1, 2]);
<immutable digraph with 9 vertices, 6 edges>
> DigraphOrbits(D);
[ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
> D := DigraphMutableCopy(D);
<mutable digraph with 9 vertices, 6 edges>
> DigraphOrbits(D);
[ [ 1, 2, 3 ], [ 4, 5, 6, 7, 8, 9 ] ]

7.2.12 DigraphOrbitReps

DigraphOrbitReps(digraph)

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 digraph.

Example

> D := CayleyDigraph(AlternatingGroup(4));
<immutable digraph with 12 vertices, 24 edges>
> DigraphOrbitReps(D);
Digraphs

7.2.13 DigraphSchreierVector

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

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 \(\text{sch}\) of integers with length \(\text{DigraphNrVertices}(\text{digraph})\) where:

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

**Example**

```gap
gap> n := 4;;
gap> adj := function(x, y)
   > return (((x - y) mod n) = 1) or (((x - y) mod n) = n - 1);
   > end;

gap> group := CyclicGroup(IsPermGroup, n);
Group([ (1,2,3,4) ])

gap> 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, 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

gap> 7 / gens[sch[7]];
5

gap> 5 / gens[sch[5]];
1
```
7.2.14 DigraphStabilizer

\[ \text{DigraphStabilizer}(\text{digraph}, v) \]  
\textbf{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 digraph.

\begin{example}
\begin{verbatim}
gap> D := DigraphFromDigraph6String("&GYHPQgWTIIPW");
<immutable digraph with 8 vertices, 24 edges>
gap> DigraphStabilizer(D, 8);
Group(())
gap> DigraphStabilizer(D, 2);
Group(())
gap> D := DigraphMutableCopy(D);
<mutable digraph with 8 vertices, 24 edges>
gap> DigraphStabilizer(D, 8);
Group(())
gap> DigraphStabilizer(D, 2);
Group(())
\end{verbatim}
\end{example}

7.2.15 IsIsomorphicDigraph (for digraphs)

\[ \text{IsIsomorphicDigraph}(\text{digraph1}, \text{digraph2}) \]  
\textbf{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).

\begin{example}
\begin{verbatim}
gap> digraph1 := CycleDigraph(4);
<immutable cycle digraph with 4 vertices>
gap> digraph2 := CycleDigraph(5);
<immutable cycle digraph with 5 vertices>
gap> IsIsomorphicDigraph(digraph1, digraph2);
false
gap> digraph2 := DigraphReverse(digraph1);
<immutable digraph with 4 vertices, 4 edges>
gap> IsIsomorphicDigraph(digraph1, digraph2);
true
gap> digraph1 := Digraph([[3], [4], [5]]);
<immutable digraph with 3 vertices, 1 edge>
gap> digraph2 := Digraph([[1], [6], [2]]);
<immutable digraph with 3 vertices, 1 edge>
gap> IsIsomorphicDigraph(digraph1, digraph2);
true
\end{verbatim}
\end{example}
### 7.2.16 IsIsomorphicDigraph (for digraphs and homogeneous lists)

**Operation:**

\[
\text{IsIsomorphicDigraph}(\text{digraph1, digraph2, colours1, colours2})
\]

**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(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 \text{IsomorphismDigraphs (7.2.18)} for more information about isomorphisms of coloured digraphs.

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

**Example**

```gap
digraph1 := ChainDigraph(4);
<immutable chain digraph with 4 vertices>
digraph2 := ChainDigraph(3);
<immutable chain digraph with 3 vertices>
IsIsomorphicDigraph(digraph1, digraph2,
[[1, 4], [2, 3]], [[1, 2], [3]]);
false
digraph2 := DigraphReverse(digraph1);
<immutable digraph with 4 vertices, 3 edges>
IsIsomorphicDigraph(digraph1, digraph2,
[[1, 1, 1, 1], [1, 1, 1, 1]]);
true
IsIsomorphicDigraph(digraph1, digraph2,
[[1, 2, 2, 1], [1, 2, 2, 1]]);
true
IsIsomorphicDigraph(digraph1, digraph2,
[[1, 1, 2, 2], [1, 1, 2, 2]]);
false
```

### 7.2.17 IsomorphismDigraphs (for digraphs)

**Operation:**

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

**Returns:** A permutation, or a pair of permutations, or fail.

This operation returns an isomorphism between the digraphs \text{digraph1} and \text{digraph2} if one exists, else this operation returns fail.

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

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

\[
\text{OnDigraphs}(\text{digraph1}, p) = \text{digraph2}.
\]

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
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
```

### 7.2.18 IsomorphismDigraphs (for digraphs and homogeneous lists)

**Operation**

\[ \text{IsomorphismDigraphs(digraph1, digraph2, colours1, colours2)} \]

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

This operation searches for an isomorphism between 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 \([1 .. m]\) for some \( m \leq n \); or
- a list of non-empty disjoint lists whose union is \( \text{DigraphVertices(digraph)} \), such that \( \text{colours}[i] \) 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 IsomorphismDigraphs (7.2.17) for more information about isomorphisms of digraphs. More precisely, let \( f \) be an isomorphism of digraphs from the digraph \( \text{digraph1} \) (with colouring \( \text{colours1} \)) to the digraph \( \text{digraph2} \) (with colouring \( \text{colours2} \)), and let \( p \) be the permutation of the vertices of \( \text{digraph1} \) that corresponds to \( f \). Then \( f \) preserves the colourings of \( \text{digraph1} \) and \( \text{digraph2} \) — and hence is an isomorphism of coloured digraphs — if \( \text{colours1}[i] = \text{colours2}[i \sim p] \) for all vertices \( i \) in \( \text{digraph1} \).

This operation returns such an isomorphism if one exists, else it returns 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).
Digraphs

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
```

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.

```
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 ] ]
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);
```
Digraphs

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)

> IsDigraphIsomorphism(src, ran, x)  
> IsDigraphIsomorphism(src, ran, x, col1, col2)  
> IsDigraphAutomorphism(digraph, x)  
> 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 ^ x, v ^ 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 (7.2.5).

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 ^ x] for all vertices i of src, for IsDigraphIsomorphism.
- col[i] = col[i ^ 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

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

gap> IsDigraphAutomorphism(src, [1, 2, 3]);
false

gap> IsDigraphAutomorphism(src, [2, 3]);
true

gap> IsDigraphAutomorphism(src, [2, 3, [2, 1, 1]]);
ture
```
7.2.21 IsDigraphColouring

\[ \text{IsDigraphColouring} \]
\[ \text{(operation)} \]
\[ \text{Returns: true or false.} \]

The operation IsDigraphColouring verifies whether or not the list \( \text{list} \) describes a proper colouring of the digraph \( \text{digraph} \).

A list \( \text{list} \) describes a proper colouring of a digraph \( \text{digraph} \) if \( \text{list} \) consists of positive integers, the length of \( \text{list} \) equals the number of vertices in \( \text{digraph} \), and for any vertices \( u, v \) of \( \text{digraph} \) if \( u \) and \( v \) are adjacent, then \( \text{list}[u] \succ \text{list}[v] \).

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

See also IsDigraphHomomorphism (7.3.10).

Example:

\[
\text{gap> D := JohnsonDigraph(5, 3);} \\
\langle\text{immutable symmetric digraph with 10 vertices, 60 edges} \rangle \\
\text{gap> IsDigraphColouring(D, [1, 2, 3, 3, 2, 1, 4, 5, 6, 7]);} \\
\text{true} \\
\text{gap> IsDigraphColouring(D, [1, 2, 3, 3, 2, 1, 2, 5, 6, 7]);} \\
\text{false} \\
\text{gap> IsDigraphColouring(D, [1, 2, 3, 3, 2, 1, 2, 5, 6, -1]);} \\
\text{false} \\
\text{gap> IsDigraphColouring(D, [1, 2, 3]);} \\
\text{false} \\
\text{gap> IsDigraphColouring(D, IdentityTransformation);} \\
\text{true}
\]
7.3 Homomorphisms of digraphs

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

7.3.1 HomomorphismDigraphsFinder

\[ \text{HomomorphismDigraphsFinder}(D_1, D_2, \text{hook}, \text{user}_\text{param}, \text{max}_\text{results}, \text{hint}, \text{injective}, \text{image}, \text{partial}_\text{map}, \text{colors}_1, \text{colors}_2[, \text{order}, \text{aut}_\text{grp}]) \]

\text{(function)}

\text{Returns:} \text{The argument user}_\text{param}.

This function finds homomorphisms from the digraph \( D_1 \) to the digraph \( D_2 \) subject to the conditions imposed by the other arguments as described below.

If \( f \) and \( g \) are homomorphisms found by \text{HomomorphismDigraphsFinder}, then \( f \) cannot be obtained from \( g \) by right multiplying by an automorphism of \( D_2 \) in \( \text{aut}_\text{grp} \).

\text{hook}

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

If \text{hook} is a function, then it must have two arguments \text{user}_\text{param} (see below) and a transformation \( t \). The function \text{hook} (\text{user}_\text{param}, t) is called every time a new homomorphism \( t \) is found by \text{HomomorphismDigraphsFinder}. If the function returns \text{true}, then \text{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 \text{HomomorphismDigraphsFinder}.

If \text{hook} is \text{fail}, then a default function is used which simply adds every new homomorphism found by \text{HomomorphismDigraphsFinder} to \text{user}_\text{param}, which must be a mutable list in this case.

\text{user}_\text{param}

If \text{hook} is a function, then \text{user}_\text{param} can be any \text{GAP} object. The object \text{user}_\text{param} is used as the first argument of the function \text{hook}. For example, \text{user}_\text{param} might be a transformation semigroup, and \text{hook} (\text{user}_\text{param}, t) might set \text{user}_\text{param} to be the closure of \text{user}_\text{param} and \( t \).

If the value of \text{hook} is \text{fail}, then the value of \text{user}_\text{param} must be a mutable list.

\text{max}_\text{results}

This argument should be a positive integer or \text{infinity}. \text{HomomorphismDigraphsFinder} will return after it has found \text{max}_\text{results} homomorphisms or the search is complete, whichever happens first.

\text{hint}

This argument should be a positive integer or \text{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.

\text{injective}

This argument should be 0, 1, or 2. If it is 2, then only embeddings are found, if it is 1, then
only injective homomorphisms are found, and if it is 0 there are no restrictions imposed by this argument.

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

**image**

This argument should be a subset of the vertices of the graph $D_2$. HomomorphismDigraphsFinder only finds homomorphisms from $D_1$ to the subgraph of $D_2$ induced by the vertices 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 $D_1$. HomomorphismDigraphsFinder only finds homomorphisms extending partial_map (if any).

**colors1**

This should be a list representing possible colours of vertices in the digraph $D_1$; see AutomorphismGroup (7.2.5) for details of the permissible values for this argument.

**colors2**

This should be a list representing possible colours of vertices in the digraph $D_2$; see AutomorphismGroup (7.2.5) for details of the permissible values for this argument.

**order**

The optional argument 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 DigraphWelshPowellOrder (7.3.16), i.e. vertices ordered from highest to lowest degree. The optional argument 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 aut_grp. If this argument is not specific, it defaults to the full automorphism group of $D_2$, which may be costly to calculate.

```
Example

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)
  > Add(user_param, t * user_param[1]);
  > end;

```
7.3.2 DigraphHomomorphism

\texttt{DigraphHomomorphism(digraph1, digraph2)} (operation)

\textbf{Returns:} A transformation, or 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.

\textbf{DigraphHomomorphism} returns a single homomorphism between \textit{digraph1} and \textit{digraph2} if it exists, otherwise it returns \textit{fail}.

Example

\begin{verbatim}
gap> gr1 := ChainDigraph(3);;
gap> gr2 := Digraph([[3, 5], [2], [3, 1], [], [4]]);
<immutable digraph with 5 vertices, 6 edges>
gap> DigraphHomomorphism(gr1, gr1);
IdentityTransformation

gap> map := DigraphHomomorphism(gr1, gr2);
Transformation([ 3, 1, 5, 4, 5 ])
gap> IsDigraphHomomorphism(gr1, gr2, map);
true
\end{verbatim}

7.3.3 HomomorphismsDigraphs

\texttt{HomomorphismsDigraphs(digraph1, digraph2)} (operation)

\texttt{HomomorphismsDigraphsRepresentatives(digraph1, digraph2)} (operation)

\textbf{Returns:} A list of transformations.

HomomorphismsDigraphsRepresentatives finds every DigraphHomomorphism (7.3.2) between \textit{digraph1} and \textit{digraph2}, up to right multiplication by an element of the AutomorphismGroup (7.2.2) of \textit{digraph2}. In other words, every homomorphism \(f\) between \textit{digraph1} and \textit{digraph2} can be written as the composition \(f = g \cdot x\), where \(g\) is one of the HomomorphismsDigraphsRepresentatives and \(x\) is an automorphism of \textit{digraph2}.

HomomorphismsDigraphs returns all homomorphisms between \textit{digraph1} and \textit{digraph2}.

Example

\begin{verbatim}
gap> gr1 := ChainDigraph(3);;
gap> gr2 := Digraph([[3, 5], [2], [3, 1], [], [4]]);
\end{verbatim}
7.3.4 DigraphMonomorphism

\>
DigraphMonomorphism(digraph1, digraph2)

(operation)

Returns: A transformation, or fail.

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

\>
Example

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

\>
MonomorphismsDigraphs(digraph1, digraph2)

(operation)

\>
MonomorphismsDigraphsRepresentatives(digraph1, digraph2)

(operation)

Returns: A list of transformations.

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

\>
Example

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

7.3.6 DigraphEpimorphism

\>
DigraphEpimorphism(digraph1, digraph2)

(operation)

Returns: A transformation, or fail.

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

Example

```gap
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 ] )
```

7.3.7 EpimorphismsDigraphs

- EpimorphismsDigraphs(digraph1, digraph2) (operation)
- EpimorphismsDigraphsRepresentatives(digraph1, digraph2) (operation)

Returns: A list of transformations.

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

Example

```gap
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, 3, 2, 1 ] ), Transformation( [ 1, 3, 2, 3 ] ),
  Transformation( [ 2, 1, 2, 3 ] ), Transformation( [ 2, 1, 3, 1 ] ),
  Transformation( [ 2, 1, 3, 2 ] ), Transformation( [ 2, 3, 1, 2 ] ),
  Transformation( [ 2, 3, 1, 3 ] ), Transformation( [ 2, 3, 2, 1 ] ),
  Transformation( [ 3, 1, 2, 1 ] ), Transformation( [ 3, 1, 2, 3 ] ),
  Transformation( [ 3, 1, 3, 2 ] ), Transformation( [ 3, 2, 1, 2 ] ),
  Transformation( [ 3, 2, 1, 3 ] ), Transformation( [ 3, 2, 3, 1 ] ) ]
```

7.3.8 DigraphEmbedding

- DigraphEmbedding(digraph1, digraph2) (operation)

Returns: A transformation, or fail.

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

In other words, an embedding t is an isomorphism from digraph1 to the InducedSubdigraph (3.3.3) of digraph2 on the image of t.

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

Example

\[ \text{gap} > \text{gr := ChainDigraph(3);} \]
\[ \text{<immutable chain digraph with 3 vertices>} \]
\[ \text{gap} > \text{DigraphEmbedding(gr, CompleteDigraph(4));} \]
\[ \text{fail} \]
\[ \text{gap} > \text{DigraphEmbedding(gr, Digraph([[3], [1, 4], [1], [3]]));} \]
\[ \text{Transformation( [ 2, 4, 3, 4 ] )} \]

7.3.9 EmbeddingsDigraphs

- EmbeddingsDigraphs(D1, D2) (operation)
- EmbeddingsDigraphsRepresentatives(D1, D2) (operation)

**Returns:** A list of transformations.

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

See also IsDigraphEmbedding (7.3.11).

Example

\[ \text{gap} > \text{D1 := NullDigraph(2);} \]
\[ \text{<immutable empty digraph with 2 vertices>} \]
\[ \text{gap} > \text{D2 := CycleDigraph(5);} \]
\[ \text{<immutable cycle digraph with 5 vertices>} \]
\[ \text{gap} > \text{EmbeddingsDigraphsRepresentatives(D1, D2);} \]
\[ \text{[ Transformation( [ 1, 3, 3 ] ), Transformation( [ 1, 4, 3, 4 ] ) ]} \]
\[ \text{gap} > \text{EmbeddingsDigraphs(D1, D2);} \]
\[ \text{[ Transformation( [ 1, 3, 3 ] ), Transformation( [ 1, 4, 3, 4 ] ),} \]
\[ \text{Transformation( [ 2, 4, 4, 5, 1 ] ),} \]
\[ \text{Transformation( [ 2, 5, 4, 5, 1 ] ),} \]
\[ \text{Transformation( [ 3, 1, 5, 1, 2 ] ),} \]
\[ \text{Transformation( [ 3, 5, 5, 1, 2 ] ),} \]
\[ \text{Transformation( [ 4, 1, 1, 2, 3 ] ),} \]
\[ \text{Transformation( [ 4, 2, 1, 2, 3 ] ),} \]
\[ \text{Transformation( [ 5, 2, 2, 3, 4 ] ),} \]
\[ \text{Transformation( [ 5, 3, 2, 3, 4 ] )}] \]

7.3.10 IsDigraphHomomorphism (for digraphs and a permutation or transformation)

- IsDigraphHomomorphism(src, ran, x) (operation)
- IsDigraphHomomorphism(src, ran, x, col1, col2) (operation)
- IsDigraphEpimorphism(src, ran, x) (operation)
- IsDigraphEpimorphism(src, ran, x, col1, col2) (operation)
- IsDigraphMonomorphism(src, ran, x) (operation)
- IsDigraphMonomorphism(src, ran, x, col1, col2) (operation)
- IsDigraphEndomorphism(digraph, x) (operation)
- IsDigraphEndomorphism(digraph, x, col) (operation)

**Returns:** true or false.

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

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

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:

- \([u \sim x, v \sim x]\) 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 GeneratorsOfEndomorphismMonoid (7.3.13).

If \( \text{col1} \) and \( \text{col2} \), or \( \text{col} \), are given, then they must represent vertex colourings; see AutomorphismGroup (7.2.5) for details of the permissible values for these argument. The homomorphism must then also have the property:

- \( \text{col}[i] = \text{col}[i \sim x] \) for all vertices \( i \) of \( \text{digraph} \), in the case of IsDigraphEndomorphism.

- \( \text{col1}[i] = \text{col2}[i \sim x] \) for all vertices \( i \) of \( \text{src} \), in the cases of the other operations.

See also DigraphsRespectsColouring (7.3.12).

**Example**

```
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([2, 1, 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> IsDigraphEpimorphism(src, ran, Transformation([3, 3, 3]));  
false  
gap> IsDigraphMonomorphism(src, ran, Transformation([1, 2, 2]));  
```
Digraphs

false
gap> IsDigraphEpimorphism(src, ran, Transformation([1, 2, 2]));
true
gap> IsDigraphMonomorphism(ran, src, ());
true

7.3.11  IsDigraphEmbedding (for digraphs and a permutation or transformation)

\[ \text{IsDigraphEmbedding}(\text{src}, \text{ran}, x) \]
\[ \text{IsDigraphEmbedding}(\text{src}, \text{ran}, x, \text{co11}, \text{co12}) \]

\textbf{Returns:} true or false.

IsDigraphEmbedding returns true if the permutation or transformation \( x \) is a embedding of the digraph \( \text{src} \) into the digraph \( \text{ran} \), while respecting the colourings \( \text{co11} \) and \( \text{co12} \) if given.

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

Example

\[
\begin{align*}
\text{gap} & \text{> src := Digraph([[1], [1, 2]]); } \\
& \text{<immutable digraph with 2 vertices, 3 edges}> \\
\text{gap} & \text{> ran := Digraph([[1], [1, 2], [1, 3]]); } \\
& \text{<immutable digraph with 3 vertices, 5 edges}> \\
\text{gap} & \text{> IsDigraphMonomorphism(src, ran, ()); } \\
& \text{true} \\
\text{gap} & \text{> IsDigraphEmbedding(src, ran, ()); } \\
& \text{true} \\
\text{gap} & \text{> IsDigraphEmbedding(src, ran, (), [2, 1], [2, 1, 1]); } \\
& \text{true} \\
\text{gap} & \text{> IsDigraphEmbedding(src, ran, (), [2, 1], [1, 2, 1]); } \\
& \text{false} \\
\text{gap} & \text{> ran := Digraph([[1, 2], [1, 2], [1, 3]]); } \\
& \text{<immutable digraph with 3 vertices, 6 edges}> \\
\text{gap} & \text{> IsDigraphMonomorphism(src, ran, IdentityTransformation); } \\
& \text{true} \\
\text{gap} & \text{> IsDigraphEmbedding(src, ran, IdentityTransformation); } \\
& \text{false}
\end{align*}
\]

7.3.12  DigraphsRespectsColouring

\[ \text{DigraphsRespectsColouring}(\text{src}, \text{ran}, x, \text{co11}, \text{co12}) \]

\textbf{Returns:} true or false.

The operation DigraphsRespectsColouring verifies whether or not the permutation or transformation \( x \) respects the vertex colourings \( \text{co11} \) and \( \text{co12} \) of the digraphs \( \text{src} \) and range. That is, DigraphsRespectsColouring returns true if and only if for all vertices \( i \) of \( \text{src} \), \( \text{co11}[i] = \text{co12}[i \ ^x] \).

Example

\[
\begin{align*}
\text{gap} & \text{> src := Digraph([[1], [1, 2]]); } \\
& \text{<immutable digraph with 2 vertices, 3 edges}> \\
\text{gap} & \text{> ran := Digraph([[1], [1, 2], [1, 3]]); } \\
& \text{<immutable digraph with 3 vertices, 5 edges>}
\end{align*}
\]
7.3.13 GeneratorsOfEndomorphismMonoid

\[ \text{GeneratorsOfEndomorphismMonoid}(\text{digraph}[, \text{colors}][, \text{limit}]) \]

- GeneratorsOfEndomorphismMonoid(digraph[, colors][, limit]) (function)
- GeneratorsOfEndomorphismMonoidAttr(digraph) (attribute)

**Returns:** A list of transformations.

An endomorphism of \textit{digraph} is a homomorphism \textit{DigraphHomomorphism} (7.3.2) from \textit{digraph} back to itself. \textit{GeneratorsOfEndomorphismMonoid}, called with a single argument, returns a generating set for the monoid of all endomorphisms of \textit{digraph}. If \textit{digraph} belongs to \textit{IsImmutableDigraph} (3.1.3), then the value of \textit{GeneratorsOfEndomorphismMonoid} will not be recomputed on future calls.

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

- A list of positive integers of size the number of vertices of \textit{digraph}, where \textit{colors} \[ i \] is the colour of vertex \( i \).
- A list of lists, such that \textit{colors} \[ i \] is a list of all vertices with colour \( i \).

If the \textit{limit} argument is specified, then it will return only the first \textit{limit} homomorphisms, where \textit{limit} must be a positive integer or infinity.

**Example**

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

7.3.14 DigraphColouring (for a digraph and a number of colours)

\[ \text{DigraphColouring(digraph, n)} \]

- DigraphColouring(digraph, n) (operation)

**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

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

7.3.15 DigraphGreedyColouring (for a digraph and vertex order)

▷ DigraphGreedyColouring(digraph, order)  
▷ DigraphGreedyColouring(digraph, func)  
▷ DigraphGreedyColouring(digraph) 

Returns: A transformation, or fail.

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

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

If digraph is a digraph and func is a function whose argument is a digraph, and that returns a dense list order, then DigraphGreedyColouring(digraph, func) returns DigraphGreedyColouring(digraph, func(digraph)).

If the optional second argument (either a list or a function), is not specified, then DigraphWelshPowellOrder (7.3.16) is used by default.

See also DigraphColouring (7.3.14).

Example

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

- **DigraphWelshPowellOrder(digraph)** (attribute)

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

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

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

7.3.17 **ChromaticNumber**

- **ChromaticNumber(digraph)** (attribute)

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

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

  If `digraph` is a digraph without loops (see `DigraphHasLoops` (6.1.1), then `ChromaticNumber` returns the least non-negative integer `n` such that there is a proper colouring of `digraph` with `n` colours. In other words, for a digraph with at least one vertex, `ChromaticNumber` returns the least number `n` such that `DigraphColouring(digraph, n)` does not return `fail`. See `DigraphColouring` (7.3.14).

  ```
  gap> ChromaticNumber(NullDigraph(10)); 1
  gap> ChromaticNumber(CompleteDigraph(10)); 10
  gap> ChromaticNumber(CompleteBipartiteDigraph(5, 5)); 2
  gap> ChromaticNumber(Digraph([[], [3], [5], [2, 3], [4]])); 3
  gap> ChromaticNumber(NullDigraph(0)); 0
  gap> D := PetersenGraph(IsMutableDigraph);
  <mutable digraph with 10 vertices, 30 edges>
  gap> ChromaticNumber(D); 3
  ```

7.3.18 **DigraphCore**

- **DigraphCore(D)** (attribute)

  **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

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 ]
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 \textit{digraph} are equal to the independent sets of the symmetric digraph:

- \textbf{DigraphSymmetricClosure}(\textbf{DigraphRemoveLoops}(\textbf{DigraphRemoveAllMultipleEdges}(\textit{digraph}))).

Again, the automorphism group of this symmetric digraph contains the automorphism group of \textit{digraph}. Since a search for independent sets is equal to a search for cliques in the DigraphDual (3.3.10), the methods used in \textbf{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 \textit{digraph} will return orbit representatives of independent sets under the action of the automorphism group of the \textit{symmetric closure} of the digraph formed by removing loops and multiple edges.

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

### 8.1 Finding cliques

#### 8.1.1 IsClique

\begin{verbatim}
\textbf{IsClique}(\textit{digraph}, \textit{l}) \quad \text{(operation)}
\end{verbatim}

\begin{verbatim}
\textbf{IsMaximalClique}(\textit{digraph}, \textit{l}) \quad \text{(operation)}
\end{verbatim}

\textbf{Returns}: true or false.

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

A \textit{clique} of a digraph is a set of mutually adjacent vertices of the digraph. A \textit{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.

\begin{verbatim}
\textbf{gap> }D := \text{CompleteDigraph}(4);;
\textbf{gap> }\text{IsClique}(D, [1, 3, 2]);
true
\textbf{gap> }\text{IsMaximalClique}(D, [1, 3, 2]);
false
\textbf{gap> }\text{IsMaximalClique}(D, \text{DigraphVertices}(D));
true
\textbf{gap> }D := \text{Digraph}([[1, 2, 4, 4], [1, 3, 4], [2, 1], [1, 2]]);
\textbf{<immutable multidigraph with 4 vertices, 11 edges>}
\textbf{gap> }\text{IsClique}(D, [2, 3, 4]);
false
\textbf{gap> }\text{IsMaximalClique}(D, [1, 2, 4]);
true
\textbf{gap> }D := \text{CompleteDigraph}(!\text{IsMutableDigraph}, 4);;
\textbf{gap> }\text{IsClique}(D, [1, 3, 2]);
true
\end{verbatim}
8.1.2 CliquesFinder

Digraphs

> CliquesFinder(digraph, hook, user_param, limit, include, exclude, max, size, reps)

Returns: The argument user_param.

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

Let $G$ denote the automorphism group of the maximal symmetric subdigraph of digraph without loops (see AutomorphismGroup (7.2.2) and MaximalSymmetricSubdigraphWithoutLoops (3.3.5)).

hook
This argument should be a function or fail.

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

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

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

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

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

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

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

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

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

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

reps
This argument should be true or false.
If \( \text{reps} \) 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 \( \text{reps} \) 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.

Example

```
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, fail, [], 5, [2, 4], [3], false, fail, false);  
[ [ 2, 4 ], [ 1, 2, 4 ], [ 2, 4, 5 ], [ 1, 2, 4, 5 ] ]  
gap> CliquesFinder(D, fail, [], 2, [2, 4], [3], false, fail, false);  
[ [ 2, 4 ], [ 1, 2, 4 ] ]  
gap> CliquesFinder(D, fail, [], infinity, [], [], true, 5, false);  
[ [ 1, 2, 3, 4, 5 ] ]  
gap> CliquesFinder(D, fail, [], infinity, [1, 3], [], false, 3, false);  
[ [ 1, 2, 3 ], [ 1, 3, 4 ], [ 1, 3, 5 ] ]  
gap> CliquesFinder(D, fail, [], 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, fail, [], 5, [2, 4], [3], false, fail, false);  
[ [ 2, 4 ], [ 1, 2, 4 ], [ 2, 4, 5 ], [ 1, 2, 4, 5 ] ]  
gap> CliquesFinder(D, fail, [], 2, [2, 4], [3], false, fail, false);  
[ [ 2, 4 ], [ 1, 2, 4 ] ]  
gap> CliquesFinder(D, fail, [], infinity, [], [], true, 5, false);  
[ [ 1, 2, 3, 4, 5 ] ]  
gap> CliquesFinder(D, fail, [], infinity, [1, 3], [], false, 3, false);  
[ [ 1, 2, 3 ], [ 1, 3, 4 ], [ 1, 3, 5 ] ]  
gap> CliquesFinder(D, fail, [], infinity, [1, 3], [], true, 3, false);  
[ ]
```
8.1.3 DigraphClique

DigraphClique(digraph[, include[, exclude[, size]]])

DigraphMaximalClique(digraph[, include[, exclude[, size]]])

Returns: An immutable list of positive integers, or fail.

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

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

One or two arguments

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

Three arguments

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

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

Four arguments

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

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

```
Example

gap> D := Digraph([[2, 3, 4], [1, 3], [1, 2], [1, 5], []]);
<immutable digraph with 5 vertices, 9 edges>
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]);
```

Digraphs

[ 1, 4 ]
gap> DigraphMaximalClique(D, [1], [3, 4]);
fail

gap> DigraphClique(D, [1, 5]);
fail

gap> DigraphClique(D, [], [], 2);
[ 1, 2 ]
gap> D := Digraph(IsMutableDigraph,
> [[2, 3, 4], [1, 3], [1, 2], [1, 5], []]);
<mutable digraph with 5 vertices, 9 edges>
gap> IsSymmetricDigraph(D);
false
gap> DigraphClique(D);
[ 5 ]

8.1.4 DigraphMaximalClique

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

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

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

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

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

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

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 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 DigraphMaximalCliquesReps and 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>

D := Digraph(CompleteDigraph(4));

G := AutomorphismGroup(D);
Group([ (5,6), (1,2), (1,5)(2,6)(3,4) ])

G := AutomorphismGroup(D);
Group([ (5,6), (1,2), (1,5)(2,6)(3,4) ])

Group([ (5,6), (1,2), (1,5)(2,6)(3,4) ])

D := Digraph(IsMutableDigraph, [2, 3], [1, 3], [1, 2, 4], [3, 5, 6], [4, 6], [4, 5]);
"mutable digraph with 6 vertices, 14 edges"

Example

8.1.5 CliqueNumber

$\Delta$

CliqueNumber(digraph)

Returns: A non-negative integer.

If $\text{digraph}$ is a digraph, then 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.

```
8.2 Finding independent sets

8.2.1 IsIndependentSet

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

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

\textbf{Returns:} true or false.

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

An independent set of a digraph is a set of mutually non-adjacent vertices of the digraph. A maximal independent set is an independent set 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.

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

8.2.2 DigraphIndependentSet

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

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

\textbf{Returns:} An immutable list of positive integers, or fail.

If digraph is a digraph, then these functions returns an independent set of 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 include and exclude must each be a (possibly empty) duplicate-free list of vertices of digraph, and the optional argument size must be a positive integer. By default,
include and exclude are empty. These functions will search for an independent set of digraph 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. This independent set may not be maximal.

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

**Example**

```gap
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>
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 ]

8.2.3 DigraphMaximalIndependentSets

\[ \text{DigraphMaximalIndependentSets}(\text{digraph}[, \text{include}[, \text{exclude}[, \text{limit}[, \text{size}]]]]) \] (function)
\[ \text{DigraphMaximalIndependentSetsReps}(\text{digraph}[, \text{include}[, \text{exclude}[, \text{limit}[, \text{size}]]]]) \] (function)
\[ \text{DigraphIndependentSets}(\text{digraph}[, \text{include}[, \text{exclude}[, \text{limit}[, \text{size}]]]]) \] (function)
\[ \text{DigraphIndependentSetsReps}(\text{digraph}[, \text{include}[, \text{exclude}[, \text{limit}[, \text{size}]]]]) \] (function)
\[ \text{DigraphMaximalIndependentSetsAttr}(\text{digraph}) \] (attribute)
\[ \text{DigraphMaximalIndependentSetsRepsAttr}(\text{digraph}) \] (attribute)

**Returns:** An immutable list of lists of positive integers.

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

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

The functions will return as many suitable independent sets as possible, up to the number \text{limit}. These functions will find independent sets that contain all of the vertices of \text{include} but do not contain any of the vertices of \text{exclude} The argument \text{size} restricts the search to those cliques that contain precisely \text{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 \text{G} denote the \text{AutomorphismGroup} (7.2.2) of the \text{DigraphSymmetricClosure} (3.3.11) of the digraph formed from \text{digraph} by removing loops and ignoring the multiplicity of edges.
Distinct independent sets

DigraphMaximalIndependentSets and DigraphIndependentSets each return a duplicate-free list of at most \textit{limit} independent sets of \textit{digraph} that satisfy the arguments.

The computation may be significantly faster if \textit{include} and \textit{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 \textit{include} and \textit{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 \textit{limit} orbits representatives (under the action of \(G\) on sets of vertices) of independent sets of \textit{digraph} that satisfy the arguments.

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

Example

```gap
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 ], [ 1, 4 ] ]
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 ], [ 1, 4 ] ]
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

\> \texttt{Splash(str[, opts])}

\textbf{Returns:} Nothing.

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

The string \texttt{str} must correspond to a valid dot or LaTeX text file and you must have have GraphViz and pdflatex installed on your computer. For details about these file formats, see \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 \texttt{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} (\textbf{Reference: 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 GraphViz or pdflatex.

\textbf{type} this option can be used to specify that the string \texttt{str} contains a LaTeX or 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}
\textbf{DotDigraph}(digraph)
\textbf{DotVertexLabelledDigraph}(digraph)
\end{verbatim}

\textbf{Returns:} A string.

DotDigraph produces a graphical representation of the digraph \textit{digraph}. Vertices are displayed as circles, numbered consistently with \textit{digraph}. Edges are displayed as arrowed lines between vertices, with the arrowhead of each line pointing towards the range of the edge.

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

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

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

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}
\textbf{DotSymmetricDigraph}(digraph)
\end{verbatim}

\textbf{Returns:} A string.

This function produces a graphical representation of the symmetric digraph \textit{digraph}. DotSymmetricDigraph will return an error if \textit{digraph} is not a symmetric digraph. See IsSymmetricDigraph (6.1.12).
Vertices are displayed as circles, numbered consistently with `digraph`. Since `digraph` is symmetric, for every non-loop edge there is a complementary edge with opposite source and range. `DotSymmetricDigraph` displays each pair of complementary edges as a single line between the relevant vertices, with no arrowhead.

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

The string returned by `DotSymmetricDigraph` can be written to a file using the command `FileString`. Example

```
gap> star := Digraph([[2, 2, 3, 4], [1, 1], [1], [1, 4]]);
gap> IsSymmetricDigraph(star);
gap> FileString("dot/star.dot", DotSymmetricDigraph(gr));
```

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> poset := Digraph([[1, 4], [2], [2, 3, 4], [4]]);
gap> IsPartialOrderDigraph(gr);
gap> FileString("dot/poset.dot", DotPartialOrderDigraph(gr));
```

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 $IsPartialOrderDigraph$ \((6.1.16)\). Thus every vertex of $Q$ corresponds to a strongly connected component of $digraph$. The output of $DotPreorderDigraph$ displays the $DigraphReflexiveTransitiveReduction$ \((3.3.13)\) 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 [http://www.graphviz.org](http://www.graphviz.org).

The string returned by $DotPreorderDigraph$ can be written to a file using the command $FileString$ (GAPDoc: $FileString$).

Example

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

9.1.6 DotHighlightedDigraph

> DotHighlightedDigraph($digraph$, $verts[, colour1, colour2]$) (operation)

**Returns:** A string.

$DotHighlightedDigraph$ produces a graphical representation of the digraph $digraph$, where the vertices in the list $verts$, and edges between them, are drawn with colour $colour1$ and all other vertices and edges in $digraph$ are drawn with colour $colour2$. If $colour1$ and $colour2$ are not given then $DotHighlightedDigraph$ uses black and grey respectively.

Note that $DotHighlightedDigraph$ does not validate the colours $colour1$ and $colour2$ - consult the GraphViz documentation to see what is available. See $DotDigraph$ \((9.1.2)\) for more details on the output.

Example

```
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"));
gap> DotHighlightedDigraph(digraph, [1, 2], "red", "black"));
```

9.2 Reading and writing digraphs to a file

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

**Graph6**

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

**Sparse6**

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

\textbf{Digraph6}

\textit{Digraph6} is a new format based on \textit{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.

\textbf{DiSparse6}

\textit{DiSparse6} is a new format based on \textit{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 \textit{Sparse6} format with a separation symbol in between.

\subsection{String}

\begin{verbatim}
> String(digraph)                      (attribute)
> PrintString(digraph)                 (operation)
\end{verbatim}

\textbf{Returns:} A string.

Returns a string string such that \textit{EvalString(string)} is equal to \textit{digraph}, and has the same mutability. See \textit{EvalString} (Reference: \textit{EvalString}).

The methods installed for \textit{String} make some attempts to ensure that \textit{string} has as short a length as possible, but there may exist shorter strings that also evaluate to \textit{digraph}.

It is possible that \textit{string} may contain escaped special characters. To obtain a representation of \textit{digraph} that can be entered as GAP input, please use \textit{Print} (Reference: \textit{Print}). Note that \textit{Print} for a digraph delegates to \textit{PrintString}, which delegates to \textit{String}.

\begin{verbatim}
gap> D := CycleDigraph(3);
<immutable cycle digraph with 3 vertices>
gap> Print(D);
CycleDigraph(3);
gap> G := PetersenGraph(IsMutableDigraph);
<mutable digraph with 10 vertices, 30 edges>
gap> String(G);
"DigraphFromGraph6String(IsMutableDigraph, \"IheA@GUAo\");" 
gap> Print(last);
DigraphFromGraph6String(IsMutableDigraph, "IheA@GUAo");
gap> DigraphFromGraph6String(IsMutableDigraph, "IheA@GUAo");
<mutable digraph with 10 vertices, 30 edges>
\end{verbatim}

\subsection{DigraphFromGraph6String}

\begin{verbatim}
> DigraphFromGraph6String([filt, ]str)           (operation)
> DigraphFromDigraph6String([filt, ]str)        (operation)
> DigraphFromSparse6String([filt, ]str)         (operation)
> DigraphFromDiSparse6String([filt, ]str)       (operation)
\end{verbatim}

\textbf{Returns:} A digraph.
If \texttt{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 \texttt{filt}, which should be either \texttt{IsMutableDigraph} (3.1.2) or \texttt{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.

\begin{verbatim}
Example
\end{verbatim}

\begin{verbatim}
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> DigraphFromGraph6String("k?"");
<immutable empty digraph with 0 vertices>
gap> DigraphFromGraph6String(IsMutableDigraph, "&DOOOW?"));
<mutable digraph with 5 vertices, 5 edges>
gap> DigraphFromGraph6String("&CQFG");
<immutable digraph with 4 vertices, 6 edges>
gap> DigraphFromGraph6String("&IM[SrKLc~lhesbU[F_");
<immutable digraph with 10 vertices, 51 edges>
gap> DigraphFromDiSparse6String(".CaWBGA?b");
<immutable multidigraph with 4 vertices, 9 edges>
\end{verbatim}

\subsection{Graph6String}

\begin{verbatim}
Graph6String(digraph) \hspace{1cm} (operation)
Digraph6String(digraph) \hspace{1cm} (operation)
Sparse6String(digraph) \hspace{1cm} (operation)
DiSparse6String(digraph) \hspace{1cm} (operation)
\end{verbatim}

\textbf{Returns:} A string.

These four functions return a highly compressed string fully describing the digraph \texttt{digraph}.

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

See \texttt{WriteDigraphs} (9.2.6).

\begin{verbatim}
Example
\end{verbatim}

\begin{verbatim}
Example
\end{verbatim}

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

\subsection{DigraphFile}

\begin{verbatim}
DigraphFile(filename[, coder][, mode]) \hspace{1cm} (function)
\end{verbatim}

\textbf{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 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` (9.2.5) or `WriteDigraphs` (9.2.6). The purpose of this is to reduce the overhead of recreating the file object inside the functions `ReadDigraphs` (9.2.5) or `WriteDigraphs` (9.2.6) 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` (9.2.13) 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.
**Digraphs**

**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>
```

### 9.2.5 ReadDigraphs

**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` (9.2.4), then `ReadDigraphs` returns the digraphs encoded in the file as a list. Note that if `filename` is a compressed file, which has been compressed appropriately to give a filename extension of `.gz`, `.bz2`, or `.xz`, then `ReadDigraphs` can read `filename` without it first needing to be decompressed.

If the optional argument `decoder` is specified and is a function which decodes a string into a digraph, then `ReadDigraphs` will use `decoder` to decode the digraphs contained in `filename`.

If the optional argument `n` is specified, then `ReadDigraphs` returns the `n`th digraph encoded in the file `filename`.

If the optional argument `decoder` is not specified, then `ReadDigraphs` will deduce which decoder to use based on the filename extension of `filename` (after removing the compression-related filename extensions `.gz`, `.bz2`, and `.xz`). For example, if the filename extension is `.g6`, then `ReadDigraphs` will use the graph6 decoder `DigraphFromGraph6String` (9.2.2).

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 (9.2.13) 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

\[
\begin{align*}
\text{gap} & \text{> ReadDigraphs(}
\text{> Concatenation(DIGRAPHS_Dir(), "/data/graph5.g6.gz"), 10);} \\
\text{> ReadDigraphs(}
\text{> Concatenation(DIGRAPHS_Dir(), "/data/graph5.g6.gz"), 17);} \\
\text{> ReadDigraphs(}
\text{> Concatenation(DIGRAPHS_Dir(), "/data/tree9.4.txt"));}
\end{align*}
\]

[ <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>,
<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>,
<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> ]

9.2.6 WriteDigraphs

\[
\text{WriteDigraphs(filename, digraphs[, encoder][, mode])}
\]

If digraphs is a list of digraphs or a digraph and filename is a string or an IO file object created using DigraphFile (9.2.4), 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 (9.2.13) 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> 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> ]
```
9.2.7 IteratorFromDigraphFile

> IteratorFromDigraphFile(filename[, decoder])  
> Returns: An iterator.

If filename is a string representing the name of a file containing encoded digraphs, then IteratorFromDigraphFile returns an iterator for which the value of NextIterator (Reference: NextIterator) is the next digraph encoded in the file.

If the optional argument decoder is specified and is a function which decodes a string into a digraph, then IteratorFromDigraphFile will use decoder to decode the digraphs contained in filename.

The purpose of this function is to easily allow looping over digraphs encoded in a file when loading all of the encoded digraphs would require too much memory.

To see what file types are available, see WriteDigraphs (9.2.6).

Example

```gap
> filename := Concatenation(DIGRAPHS_Dir(), "/tst/out/man.d6.gz");;
> file := DigraphFile(filename, "w");;
> for i in [1 .. 10] do
> > WriteDigraphs(file, Digraph([[1, 3], [2], [1, 2]]));
> od;
> IO_Close(file);
> iter := IteratorFromDigraphFile(filename);
> for x in iter do od;
```

9.2.8 DigraphPlainTextLineEncoder

> DigraphPlainTextLineEncoder(delimiter1[, delimiter2], offset)  
> DigraphPlainTextLineDecoder(delimiter1[, delimiter2], offset)  
> Returns: A string.

These two functions return a function which encodes or decodes a digraph in a plain text format. DigraphPlainTextLineEncoder returns a function which takes a single digraph as an argument. The function returns a string describing the edges of that digraph; each edge is written as a pair of integers separated by the string delimiter2, and the edges themselves are separated by the string delimiter1. DigraphPlainTextLineDecoder returns the corresponding decoder function, which takes a string argument in this format and returns a digraph.

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

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

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

Example

```gap
> gr := Digraph([[2, 3], [1], [1]]);
> enc := DigraphPlainTextLineEncoder(" ", ", ", -1);
> dec := DigraphPlainTextLineDecoder(" ", ", ", 1);
> enc(gr);
"0 1 0 2 1 0 2 0"
```
9.2.9 TournamentLineDecoder

> TournamentLineDecoder(str)

**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 ] ]
```

9.2.10 AdjacencyMatrixUpperTriangleLineDecoder

> AdjacencyMatrixUpperTriangleLineDecoder(str)

**Returns:** A digraph.

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

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

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

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

Example

```gap
gap> gr := AdjacencyMatrixUpperTriangleLineDecoder("100001");
<immutable digraph with 4 vertices, 6 edges>
gap> OutNeighbours(gr);
[ [ 2 ], [ ], [ 1, 2, 4 ], [ 1, 2 ] ]
gap> gr := AdjacencyMatrixUpperTriangleLineDecoder("111111x111");
<immutable digraph with 5 vertices, 9 edges>
```
9.2.11 TCodeDecoder

\textbf{TCodeDecoder} (\texttt{str})

\textbf{Returns:} A digraph.

If \texttt{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 \([0 \ldots v-1]\). These integers are read in pairs and define the digraph’s edges. This function will return an error if \texttt{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 \([1,7]\).

\begin{verbatim}
> gr := TCodeDecoder("3 2 0 2 2 1");
<immutable digraph with 3 vertices, 2 edges>
> OutNeighbours(gr);
[ [ 3 ], [ ], [ 2 ] ]
> gr := TCodeDecoder("12 3 0 10 5 2 8 8");
<immutable digraph with 12 vertices, 3 edges>
> OutNeighbours(gr);
[ [ 11 ], [ ], [ ], [ ], [ 3 ], [ ], [ 9 ], [ ], [ ], [ ], [ ] ]
\end{verbatim}

9.2.12 PlainTextString

\textbf{PlainTextString} (\texttt{digraph})

\textbf{DigraphFromPlainTextString} (\texttt{s})

\textbf{Returns:} A string.

\texttt{PlainTextString} takes a single digraph, and returns a string describing the edges of that digraph. \texttt{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 \texttt{DigraphFromPlainTextString} takes an optional first argument \texttt{IsMutableDigraph} (3.1.2) or \texttt{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.

\begin{verbatim}
> gr := Digraph([[2, 3], [1], [1]]);
<immutable digraph with 3 vertices, 4 edges>
> PlainTextString(gr);
"0 1 0 2 1 0 2 0"
> DigraphFromPlainTextString(last);
<immutable digraph with 3 vertices, 4 edges>
\end{verbatim}
9.2.13 WritePlainTextDigraph

\[\text{WritePlainTextDigraph}(\text{filename}, \text{digraph}, \text{delimiter}, \text{offset})\] (function)

\[\text{ReadPlainTextDigraph}(\text{filename}, \text{delimiter}, \text{offset}, \text{ignore})\] (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 \text{delimiter}.

\text{filename} should be the name of a file which will be written to or read from, and \text{offset} should be an integer which is added to each vertex number as it is written or read. For example, if \text{WritePlainTextDigraph} is called with \text{offset} -1, then the vertices will be numbered in the file starting from 0 instead of 1 - \text{ReadPlainTextDigraph} would then need to be called with \text{offset} 1 to convert back to the original graph.

\text{ignore} should be a list of characters which will be ignored when reading the graph.

Example

```
gap> gr := Digraph([[1, 2, 3], [1, 1], [2]]);  
<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, ['/','%']);
```

9.2.14 WriteDIMACSDigraph

\[\text{WriteDIMACSDigraph}(\text{filename}, \text{digraph})\] (operation)

\[\text{ReadDIMACSDigraph}(\text{filename})\] (operation)

These operations write or read the single symmetric digraph \text{digraph} to or from a file in DIMACS format, as appropriate. The operation \text{WriteDIMACSDigraph} records the vertices and edges of \text{digraph}. The vertex labels of \text{digraph} will be recorded only if they are integers. See \text{IsSymmetricDigraph} (6.1.12) and \text{DigraphVertexLabels} (5.1.9).

The first argument \text{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 \text{filename} ends in one of .gz, .bz2, or .xz, then the file is compressed, or decompressed, appropriately.

The DIMACS format is described as follows. Each line in the DIMACS file has one of four types:

- A line beginning with c and followed by any number of characters is a comment line, and is ignored.

- A line beginning with p defines the numbers of vertices and edges the digraph. This line has the format <nr_vertices> <nr_edges>, where <nr_vertices> and <nr_edges> are replaced by the relevant integers. There must be exactly one such line in the file, and it must occur before any of the following kinds of line.

Although it is required to be present, the value of <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 e <c> <u>, where <v> and <u> are integers in the range [1 .. <nr_vertices>], specifies that there is a (symmetric) edge in the digraph between the ver-
\begin{verbatim}
Example
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}

Vertices \textless v \textgreater and \textless w \textgreater. A symmetric edge only needs to be defined once; an additional line \textless e \textless v \textgreater <w>, or \textless e \textless w \textgreater <v>, will be interpreted as an additional, multiple edge. Loops are permitted.

- A line of the form \textless n \textless v \textgreater <label>, where \textless v \textgreater is an integer in the range \([1 \ldots <nr\_vertices>]\) and \textless label \textgreater is an integer, signifies that the vertex \textless v \textgreater has the label \textless label \textgreater in the digraph. If a label is not specified for a vertex, then \texttt{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.
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 (5.3.25)</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, 63
AdjacencyMatrixMutableCopy, 63
AdjacencyMatrixUpperTriangleLineDecoder, 171
ArticulationPoints, 80
AsBinaryRelation, 17
AsDigraph, 17
AsGraph, 19
AsMonoid, 91
AsSemigroup
  for a filter and a digraph, 91
  for a filter, semilattice digraph, and two lists, 92
AsTransformation, 19
AutomorphismGroup
  for a digraph, 122
  for a digraph and a homogeneous list, 123
  for a digraph, homogeneous list, and list, 124
BipartiteDoubleDigraph, 42
BlissAutomorphismGroup
  for a digraph, 122
  for a digraph and homogenous list, 122
  for a digraph, homogenous list, and list, 122
BlissCanonicalDigraph, 127
BlissCanonicalLabelling
  for a digraph, 125
  for a digraph and a list, 126
BooleanAdjacencyMatrix, 64
BooleanAdjacencyMatrixMutableCopy, 64
Bridges, 81
CayleyDigraph, 16
ChainDigraph, 47
CharacteristicPolynomial, 64
ChromaticNumber, 147
CliqueNumber, 155

CliqueFinder, 151
CompleteBipartiteDigraph, 47
CompleteDigraph, 47
CompleteMultipartiteDigraph, 48
CycleDigraph, 48
DegreeMatrix, 71
Digraph, 12
  for a group, list, function, and function, 12
  for a list and function, 12
Digraph6String, 165
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, 65
DigraphAllSimpleCircuits, 86
DigraphBicomponents, 80
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, 153
DigraphCliques, 154
DigraphCliquesReps, 154
DigraphClosure, 44
DigraphColouring
Digraphs

for a digraph and a number of colours, 145
DigraphConnectedComponent, 78
DigraphConnectedComponents, 77
DigraphCopy, 20
DigraphCopySameMutability, 19
DigraphCore, 147
DigraphDegeneracy, 87
DigraphDegeneracyOrdering, 88
DigraphDiameter, 73
DigraphDijkstra
  for a source, 90
  for a source and target, 90
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, 75
  for a digraph, a pos int, and an int, 75
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, 141
DigraphEpiomorphism, 140
DigraphFamily, 12
DigraphFile, 165
DigraphFloydWarshall, 83
DigraphFromDigraph6String, 164
DigraphFromDiSparse6String, 164
DigraphFromGraph6String, 164
DigraphFromPlainTextString, 172
DigraphFromSparse6String, 164
DigraphGirth, 75
DigraphGreedyColouring
  for a digraph, 146
  for a digraph and vertex order, 146
  for a digraph and vertex order function, 146
DigraphGroup, 128
DigraphHasLoops, 98
DigraphHomomorphism, 139
DigraphImmutableCopy, 19
DigraphImmutableCopyIfImmutable, 20
DigraphImmutableCopyIfMutable, 20
DigraphIndependentSet, 156
DigraphIndependentSets, 158
DigraphIndependentSetsReps, 158
DigraphInEdges, 60
DigraphJoin
  for a list of digraphs, 38
  for a positive number of digraphs, 38
DigraphLayers, 87
DigraphLongestDistanceFromVertex, 74
 DigraphLongestSimpleCircuit, 87
DigraphLoops, 70
DigraphMaximalClique, 153
DigraphMaximalCliques, 154
DigraphMaximalCliquesAttr, 154
DigraphMaximalCliquesReps, 154
DigraphMaximalCliquesRepsAttr, 154
DigraphMaximalIndependentSet, 156
DigraphMaximalIndependentSets, 158
DigraphMaximalIndependentSetsAttr, 158
DigraphMaximalIndependentSetsReps, 158
DigraphMaximalIndependentSetsRepsAttr, 158
DigraphMaximalMatching, 62
DigraphMaximumMatching, 62
DigraphMonomorphism, 140
DigraphMutableCopy, 19
DigraphMutableCopyIfImmutable, 20
DigraphMutableCopyIfMutable, 20
DigraphMycielskian, 44
DigraphMycielskianAttr, 44
DigraphNrConnectedComponents, 77
DigraphNrEdges, 55
DigraphNrStronglyConnectedComponents, 78
DigraphNrVertices, 54
DigraphOddGirth, 76
DigraphOrbitReps, 129
DigraphOrbits, 129
DigraphOutEdges, 60
DigraphPath, 84
DigraphPeriod, 82
DigraphPlainTextLineDecoder, 170
DigraphPlainTextLineEncoder, 170
Digraphs

InDegreeOfVertex, 70
InDegrees, 68
InDegreeSequence, 68
InDegreeSet, 68
InducedSubdigraph, 21
InNeighbors, 67
InNeighborsMutableCopy, 67
InNeighborsOfVertex, 70
InNeighbours, 67
InNeighboursMutableCopy, 67
InNeighboursOfVertex, 70
IsAcyclicDigraph, 109
IsAntiSymmetricDigraph, 98
IsAsymmetricDigraph, 98
IsAperiodicDigraph, 113
IsBiconnectedDigraph, 110
IsBipartiteDigraph, 99
IsBridgelessDigraph, 111
IsCayleyDigraph, 12
IsChainDigraph, 109
IsClique, 150
IsCompleteBipartiteDigraph, 99
IsCompleteDigraph, 100
IsCompleteMultipartiteDigraph, 100
IsConnectedDigraph, 110
IsCycleDigraph, 116
IsDigraph, 11
IsDigraphAutomorphism, 135
for a digraph and a transformation or permutation, 135
IsDigraphColouring, 136
for a transformation, 136
IsDigraphCore, 118
IsDigraphEdge
for digraph and list, 61
for digraph and two pos ints, 61
IsDigraphEmbedding, 144
for digraphs and a permutation or transformation, 144
IsDigraphEndomorphism, 142
for digraphs and a permutation or transformation, 142
IsDigraphEpimorphism, 142
for digraphs and a permutation or transformation, 142
IsDigraphIsomorphism, 135
for digraphs and transformation or permutation, 135
IsDigraphMonomorphism, 142
for digraphs and a permutation or transformation, 142
IsDigraphWithAdjacencyFunction, 12
IsDirectedTree, 113
IsDistanceRegularDigraph, 108
IsEdgeTransitive, 118
IsEmptyDigraph, 101
IsEquivalenceDigraph, 101
IsEulerianDigraph, 114
IsFunctionalDigraph, 102
IsHamiltonianDigraph, 115
IsImmutableDigraph, 11
IsIndependentSet, 156
IsInRegularDigraph, 107
IsIsomorphicDigraph
for digraphs, 131
for digraphs and homogeneous lists, 132
IsJoinSemilatticeDigraph, 106
IsLatticeDigraph, 106
IsMatching, 61
IsMaximalClique, 150
IsMaximalIndependentSet, 156
IsMaximalMatching, 61
IsMaximumMatching, 61
IsMeetSemilatticeDigraph, 106
IsMultiDigraph, 102
IsMutableDigraph, 11
IsNullDigraph, 101
IsomorphismDigraphs
for digraphs, 132
for digraphs and homogeneous lists, 133
IsOuterPlanarDigraph, 117
IsOutRegularDigraph, 107
IsPartialOrderDigraph, 106
IsPerfectMatching, 61
IsPlanarDigraph, 116
IsPreorderDigraph, 105
IsQuasiorderDigraph, 105
IsReachable, 84
IsReflexiveDigraph, 103
IsRegularDigraph, 108
IsStronglyConnectedDigraph, 112
IsSubdigraph, 51
IsSymmetricDigraph, 103
IsTournament, 104
IsTransitiveDigraph, 104
IsUndirectedForest, 114
IsUndirectedSpanningForest, 52
IsUndirectedSpanningTree, 52
IsUndirectedTree, 114
IsVertexTransitive, 118
 IteratorFromDigraphFile, 170
IteratorOfPaths, 85
JohnsonDigraph, 49
KuratowskiOuterPlanarSubdigraph, 93
KuratowskiPlanarSubdigraph, 93
LaplacianMatrix, 72
LineDigraph, 41
LineUndirectedDigraph, 42
MaximalAntiSymmetricSubdigraph, 23
MaximalAntiSymmetricSubdigraphAttr, 23
MaximalSymmetricSubdigraph, 22
MaximalSymmetricSubdigraphAttr, 22
MaximalSymmetricSubdigraphWithoutLoops, 22
MaximalSymmetricSubdigraphWithoutLoopsAttr, 22
MonomorphismsDigraphs, 140
MonomorphismsDigraphsRepresentatives, 140
NautyAutomorphismGroup, 123
NautyCanonicalDigraph, 127
NautyCanonicalLabelling
for a digraph, 125
for a digraph and a list, 126
NrSpanningTrees, 89
NullDigraph, 49
OnDigraphs
for a digraph and a perm, 120
for a digraph and a transformation, 120
OnMultiDigraphs, 121
for a digraph, perm, and perm, 121
OutDegreeOfVertex, 69
OutDegrees, 68
OutDegreeSequence, 68
OutDegreeSet, 68
OuterPlanarEmbedding, 95
OutNeighbors, 66
OutNeighborsMutableCopy, 66
OutNeighborsOfVertex, 69
OutNeighbours, 66
OutNeighboursMutableCopy, 66
OutNeighboursOfVertex, 69
PartialOrderDigraphJoinOfVertices, 71
PartialOrderDigraphMeetOfVertices, 71
PetersenGraph, 50
PlainTextString, 172
PlanarEmbedding, 94
PrintString, 164
QuotientDigraph, 25
RandomDigraph, 45
RandomLattice, 46
RandomMultiDigraph, 45
RandomTournament, 46
ReadDigraphs, 167
ReadDIMACSDigraph, 173
ReadPlainTextDigraph, 173
ReducedDigraph, 22
ReducedDigraphAttr, 22
RepresentativeOutNeighbours, 134
SemigroupOfCayleyDigraph, 90
SetDigraphEdgeLabel, 59
SetDigraphEdgeLabels
for a digraph and a function, 59
for a digraph and a list of lists, 59
SetDigraphVertexLabel, 57
SetDigraphVertexLabels, 58
Sparse6String, 165
Splash, 160
String, 164
StrongOrientation, 82
StrongOrientationAttr, 82
SubdigraphHomeomorphicToK23, 96
SubdigraphHomeomorphicToK33, 96
SubdigraphHomeomorphicToK4, 96
TCodeDecoder, 172
TournamentLineDecoder, 171

UndirectedSpanningForest, 24
UndirectedSpanningForestAttr, 24
UndirectedSpanningTree, 24
UndirectedSpanningTreeAttr, 24

WriteDigraphs, 168
WriteDIMACSDigraph, 173
WritePlainTextDigraph, 173