

NautyTracesInterface

An interface to nauty

0.3

20 March 2025

Sebastian Gutsche

Sebastian Gutsche

Email: gutsche@mathematik.uni-siegen.de

Homepage: <https://sebasguts.github.io>

Address: TODO

Contents

1	Nauty Graphs	3
1.1	Working with Nauty Graphs	3
2	Nauty Graphs with labels	9
2.1	Working with Nauty Graphs with labels	9
3	Nauty Traces Interface	12
3.1	Data Structures	12
	Index	13

Chapter 1

Nauty Graphs

1.1 Working with Nauty Graphs

1.1.1 NautyGraph (for IsList)

- ▷ `NautyGraph(edges, nr)` (operation)
- ▷ `NautyGraph(arg1, arg2)` (operation)

Returns: a `NautyGraph`

This function creates a nauty graph object for an undirected graph without multiple edges, but possibly with loops, whose edges are given by the list *edges*. The list *edges* is a list whose entries are lists of length 2, consisting of the two (possibly equal) vertices of the edges. If two edges are either equal or one is the reversed of the other, the graph created will still only have a single undirected edge. The graph created is on the vertices $1, \dots, nr$, where *nr* is the maximal entry occurring in one of the edges. If the function is called with a second argument *nr* then *nr* must be a positive integer which is at least equal to the maximal entry occurring in one of the edges.

Example

```
gap> ng := NautyGraph( [ [1,2], [2,3], [3,4], [4,1], [3,2] ] );
<An undirected Nauty graph with on 4 vertices>
gap>
[ [ 1, 2 ], [ 2, 3 ], [ 3, 4 ], [ 4, 1 ], [ 3, 2 ] ]
```

1.1.2 NautyDiGraph (for IsList)

- ▷ `NautyDiGraph(edges, nr)` (operation)
- ▷ `NautyDiGraph(arg1, arg2)` (operation)

Returns: a `NautyGraph`

This function creates a nauty graph object for a directed graph without multiple edges, but possibly with loops, whose edges are given by the list *edges*. The list *edges* is a list whose entries are lists of length 2, consisting of the two (possibly equal) vertices of the edges. If two edges are equal the graph created will still only have a single directed edge. The graph created is on the vertices $1, \dots, nr$, where *nr* is the maximal entry occurring in one of the edges. If the function is called with a second argument *nr* then *nr* must be a positive integer which is at least equal to the maximal entry occurring in one of the edges.

Example

```
gap> nautygraph := NautyDiGraph( [ [1,2],[2,3],[3,4], [4,1] ] );
<A directed Nauty graph on 4 vertices>
```

```
gap> AutomorphismGroup(nautygraph);
Group([ (1,2,3,4) ])
```

1.1.3 NautyColoredGraph (for IsList, IsList)

▷ NautyColoredGraph(*edges*, *colours*) (operation)

Returns: a NautyGraph

This function creates a nauty graph object for an undirected vertex coloured graph without multiple edges, but possibly with loops, whose edges are given by the list *edges*. The list *edges* is a list whose entries are lists of length 2, consisting of the two (possibly equal) vertices of the edges. If two edges are equal or reversed to each other the graph created will still only have a single undirected edge. The graph created is on the vertices $1, \dots, nr$, where nr is the maximal entry occurring in one of the edges. The list *colours* must be a list of length nr whose entries are positive integers. The vertex i has colour *colours*[i].

Example

```
gap> nautygraph := NautyColoredGraph( [ [1,2],[2,3],[3,4], [4,1] ], [1,2,1,2] );
<An undirected vertex-coloured Nauty graph on 4 vertices>
gap> AutomorphismGroup(nautygraph);
Group([ (2,4), (1,3) ])
```

DeclareSynonym("NautyColouredGraph", NautyColoredGraph);

1.1.4 NautyColoredDiGraph (for IsList, IsList)

▷ NautyColoredDiGraph(*edges*, *colours*) (operation)

Returns: a NautyGraph

This function creates a nauty graph object for an undirected vertex coloured graph without multiple edges, but possibly with loops, whose edges are given by the list *edges*. The list *edges* is a list whose entries are lists of length 2, consisting of the two (possibly equal) vertices of the edges. If two edges are equal or reversed to each other the graph created will still only have a single undirected edge. The graph created is on the vertices $1, \dots, nr$, where nr is the maximal entry occurring in one of the edges. The list *colours* must be a list of length nr whose entries are positive integers. The vertex i has colour *colours*[i].

Example

```
gap> nautygraph := NautyColoredDiGraph( [ [1,2],[2,3],[3,4], [4,1] ], [1,2,1,2] );
<An undirected vertex-coloured Nauty graph on 4 vertices>
gap> AutomorphismGroup(nautygraph);
Group([ (2,4), (1,3) ])
```

DeclareSynonym("NautyColouredDiGraph", NautyColoredDiGraph);

1.1.5 NautyEdgeColoredGraph (for IsList)

▷ NautyEdgeColoredGraph(*edgeclasses*, *colours*) (operation)

▷ NautyEdgeColoredGraph(*arg1*, *arg2*) (operation)

▷ NautyEdgeColoredDiGraph(*arg*) (operation)

▷ NautyEdgeColoredDiGraph(*arg1*, *arg2*) (operation)

Returns: a NautyGraph

This function creates a nauty graph object for an undirected edge coloured graph without multiple edges, but possibly with loops. The edges of the graph are specified in the argument *edgeclasses* as follows. *edgeclasses* is a list of lists L_i , where each list L_i is a list of edges, that is L_i is a list whose entries are lists of length 2, consisting of the two (possibly equal) vertices of the edges. If two edges are equal or reversed to each other the graph created will still only have a single undirected edge. The graph created is on the vertices $1, \dots, nr$, where nr is the maximal entry occurring in one of the edges. The edges in the i th list L_i have colour i .

Example

```
gap> nautygraph := NautyEdgeColoredGraph( [ [[1,2],[2,3]], [[3,4], [4,1]] ] );
<An undirected edge-coloured Nauty graph on 4 vertices and 2 edge colours>
gap> AutomorphismGroup(nautygraph);
Group([ (1,3) ])
```

```
DeclareSynonym( "NautyEdgeColouredGraph", NautyEdgeColoredGraph ); DeclareSynonym(
"NautyEdgeColouredDiGraph", NautyEdgeColoredDiGraph );
```

1.1.6 AutomorphismGroup (for IsNautyGraph)

▷ AutomorphismGroup(*graph*) (attribute)

Returns: a permutation group

Given a simple (di)graph *graph* which is a *nauty graph* object (see 1.1), this function computes the automorphism group of *graph* as a permutation group on the nodes of *graph*. As *graph* is a simple graph, its edges are given as lists $[i, j]$ of length 2 consisting of a pair of nodes in the set N . If an edge is a loop, it is of the form $[i, i]$. If i, j are different nodes and the graph is undirected, $[i, j]$ and $[j, i]$ refer to the same edge, and to different edges when the graph is directed. A bijection ϕ from N to itself is called an *automorphism* of the (undirected) graph *graph* if ϕ maps edges of *graph* to edges of *graph*, that is $e = [i, j]$ is an edge of *graph* if and only if $e = [i^\phi, j^\phi]$ or, in the undirected case, $e = [j^\phi, i^\phi]$ is an edge of *graph*. If The *automorphism group* of *graph* is returned as a permutation group acting on the set N .

Example

```
gap> nautygraph := NautyGraph( [ [1,2],[2,3],[3,4], [4,1] ] );
<An undirected Nauty graph with on 4 vertices>
gap> AutomorphismGroup(nautygraph);
Group([ (2,4), (1,2)(3,4) ])
```

1.1.7 EdgesOfNautyGraph (for IsNautyGraph)

▷ EdgesOfNautyGraph(*graph*) (attribute)

▷ Edges(*arg*) (attribute)

Returns: a list of lists of length 2 of positive integers

Given a nauty graph *graph*, this function returns the list of edges of *graph*, where an edge is a pair of vertices. If the graph is directed, the edge $[i, j]$ is the directed edge from vertex i to vertex j .

Example

```
gap> nautygraph := NautyGraph( [ [1,2],[2,3],[3,4], [4,1] ] );
<An undirected Nauty graph with on 4 vertices>
gap> EdgesOfNautyGraph(nautygraph);
[ [ 1, 2 ], [ 2, 3 ], [ 3, 4 ], [ 4, 1 ] ]
```

1.1.8 VerticesOfNautyGraph (for IsNautyGraph)

▷ VerticesOfNautyGraph(*graph*) (attribute)

Returns: a list of positive integers

Given a nauty graph *graph*, this function returns the list of vertices of *graph*.

Example

```
gap> nautygraph := NautyGraph( [ [1,2],[2,3],[3,4], [4,1] ] );
#! <An undirected Nauty graph with on 4 vertices>
gap> VerticesOfNautyGraph(nautygraph);
[ 1 .. 4 ]
```

1.1.9 VertexColoursOfNautyGraph (for IsNautyGraph)

▷ VertexColoursOfNautyGraph(*graph*) (attribute)

Returns: a list of positive integers

Given a coloured nauty graph *graph*, this function returns a list giving the colours of the vertices of *graph*.

Example

```
gap> ng := NautyColoredGraph( [ [1,2], [2,3], [3,4], [4,1], [3,2] ], [1,2,1,2] );
<An undirected vertex-coloured Nauty graph on 4 vertices>
gap> VertexColoursOfNautyGraph(ng);
[ 1, 2, 1, 2 ]
```

1.1.10 CanonicalForm (for IsNautyGraph)

▷ CanonicalForm(*graph*) (attribute)

Returns: a graph

Given a nauty graph *graph*, this function returns a graph *cangraph* which lies in the same orbit as *graph* under the automorphism group of *graph*. For the definition of which graph in the orbit is the canonical representatives see the documentation of Nauty and Traces. The computation of the canonical representative is performed by the Nauty and Traces.

Example

```
gap> ng := NautyGraph( [ [1,3], [2,3], [2,5], [4,5], [5,1] ] );
<An undirected Nauty graph with on 5 vertices>
gap> canrep := CanonicalForm(ng);
<An undirected Nauty graph with on 5 vertices>
gap> EdgesOfNautyGraph(canrep);
[ [ 1, 5 ], [ 2, 4 ], [ 2, 5 ], [ 3, 4 ], [ 3, 5 ] ]
```

1.1.11 CanonicalLabeling (for IsNautyGraph)

▷ CanonicalLabeling(*graph*) (attribute)

▷ CanonicalLabelingInverse(*graph*) (attribute)

Returns: a permutation

Given a nauty graph *graph*, the function *CanonicalLabeling* returns a permutation *perm* of the vertices of *graph* which lies in the automorphism group of *graph*. If *perm* is applied to the canonical representative of *graph* (see *CanonicalForm*), by mapping the vertices under *perm* and mapping the edges accordingly, the resulting graph is input *graph*. The function *CanonicalLabelingInverse*

returns the inverse permutation of *perm*. For the definition of the canonical representatives in the orbit of a graph under its automorphism group, see the documentation of Nauty and Traces. The computation of the canonical representative is performed by the Nauty and Traces.

Example

```
gap> ng := NautyGraph( [ [1,3], [2,3], [2,5], [4,5], [5,1] ] );
<An undirected Nauty graph with on 5 vertices>
gap> perm := CanonicalLabeling(ng);
(1,4,3,2)
gap> canrep := NautyGraph(List(Set(EdgesOfNautyGraph(ng)), i->OnTuples(i, perm^-1)));
<An undirected Nauty graph with on 5 vertices>
gap> EdgesOfNautyGraph(canrep);
[ [ 2, 4 ], [ 3, 4 ], [ 3, 5 ], [ 1, 5 ], [ 5, 2 ] ]
gap> CanonicalLabeling(ng);
(1,4,3,2)
```

1.1.12 IsomorphismGraphs (for IsNautyGraph, IsNautyGraph)

- ▷ IsomorphismGraphs(*graph*, *graph*) (operation)
- ▷ IsomorphicGraphs(*graph*, *graph*) (operation)
- ▷ IsIsomorphicGraphs(*arg1*, *arg2*) (operation)

Returns: a Permutation

Given two nauty (di)graphs *graph1* and *graph2* we say that *graph1* and *graph2* are isomorphic, if there is a bijection π from the vertices of *graph1* and to the vertices of *graph2* such that, $e = [i, j]$ is an edge of of *graph1* if and only if $e^\pi = [i^\pi, j^\pi]$ is an edge of of *graph2*. Such a bijection π is called a *graph isomorphism*. This function tests whether such a bijection π exists. If so, it returns the permutation π and otherwise fail. If in addition the nauty (di)graphs *graph1* and *graph2* are both vertex coloured, then a bijection π is additionally required to respect the partition of the node colours, that is, two nodes i, j have the same colour in *graph1* if and only if they have the same colour in *graph2*.

Example

```
gap> ng := NautyGraph([ [1,2], [1,3], [1,4], [1,5], [2,3], [2,4], [5,6], [6,7] ]);
<An undirected Nauty graph with on 7 vertices>
gap> ng2 := NautyGraph([ [1,2], [1,3], [1,4], [1,6], [2,3], [2,4], [5,6], [5,7] ]);
<An undirected Nauty graph with on 7 vertices>
gap> IsomorphismGraphs(ng, ng2);
(5,6)
```

Given two nauty (di)graphs *graph1* and *graph2* we say that *graph1* and *graph2* are isomorphic, if there is a bijection π from the vertices of *graph1* and to the vertices of *graph2* such that, $e = [i, j]$ is an edge of of *graph1* if and only if $e^\pi = [i^\pi, j^\pi]$ is an edge of of *graph2*. Such a bijection π is called a GRAPH ISOMORHISM. This function tests whether such a bijection π exists. If so, it returns true and otherwise false. If in addition the nauty (di)graphs *graph1* and *graph2* are both vertex coloured, then a bijection π is additionally required to respect the partition of the node colours, that is, two nodes i, j have the same colour in *graph1* if and only if they have the same colour in *graph2*.

Example

```
gap> ng := NautyGraph([ [1,2], [1,3], [1,4], [1,5], [2,3], [2,4], [5,6], [6,7] ]);
<An undirected Nauty graph with on 7 vertices>
gap> ng2 := NautyGraph([ [1,2], [1,3], [1,4], [1,6], [2,3], [2,4], [5,6], [5,7] ]);
```

```
<An undirected Nauty graph with on 7 vertices>  
gap> IsomorphicGraphs(ng,ng2);  
true
```


Chapter 2

Nauty Graphs with labels

2.1 Working with Nauty Graphs with labels

The package `NautyTracesInterface` allows working with graphs whose nodes are labelled. This feature is useful when working with graphs whose set of nodes is not equal to a set $\{1, \dots, n\}$ for some positive integer n . For example, consider the (di) graph on the nodes $N = \{2, 4, 6\}$ with edges $[[2, 4], [4, 6], [2, 6]]$. To work with this graph in `nauty` and `traces` directly, it is considered to be a graph with nodes $\{1, \dots, 6\}$. However, using `node labels` we can view this as a graph on three nodes, namely 1, 2, 3 and attach a label to each of these nodes. The labels are recorded in a list `labels` which defines a map from the default nodes $\{1, \dots, |N|\}$ to the set of nodes on which the edges are defined. In this example, `labels = [2, 4, 6]`. The function `NautyGraphWithNodeLabels` called with the edges $[[2, 4], [4, 6], [2, 6]]$ and labels `[2, 4, 6]` then returns a graph on 3 nodes. The graph on the default node set is called the *underlying nauty graph*.

2.1.1 NodeLabeling (for IsNautyGraphWithNodeLabels)

▷ `NodeLabeling(graph)` (attribute)

Returns: a list

Given a nauty (di)graph `graph` with node labels this function returns a dense list of positive integers, which are the node labels of `graph`. If i is a node in the underlying nauty graph, then `labels[i] = j` means that the i -th node has label j .

Example

```
gap> labels := [4,3,5,2,1];;
gap> ng := NautyDiGraphWithNodeLabels([[1,2],[1,3],[1,4],[1,5]], labels);
<An undirected Nauty graph with on 5 vertices>
gap> NodeLabeling(ng);
[ 4, 3, 5, 2, 1 ]
```

2.1.2 UnderlyingNautyGraph (for IsNautyGraphWithNodeLabels)

▷ `UnderlyingNautyGraph(graph)` (attribute)

Returns: a list

A nauty (di)graph `graph` with node labels `labels` is a nauty graph object containing an *underlying nauty graph*. The graph has a set N of nodes and edges between these

nodes. The underlying nauty graph has nodes $\{1, \dots, |N|\}$. If i is a node in the underlying nauty graph, then $labels[i] = j$ means that the i -th node has label j , where $j \in N$. Thus $labels$ is a bijection from $\{1, \dots, |N|\}$ to N . Suppose $graph$ has been constructed with `NautyDiGraphWithNodeLabels(edges, labels)`. The underlying nauty graph Γ on the nodes $\{1, \dots, nr\}$, is defined such that $e = [i, j]$ is an edge of Γ if and only if $[label[i], label[j]]$ is in the input list *edges*.

Example

```
gap> labels := [4,3,5,2,1];;
gap> ng := NautyDiGraphWithNodeLabels([[1,2],[1,3],[1,4],[1,5]], labels);
<An undirected Nauty graph with on 5 vertices>
gap> NodeLabeling(ng);
[ 4, 3, 5, 2, 1 ]
EdgesOfNautyGraph(UnderlyingNautyGraph(ng));
[ [ 5, 4 ], [ 5, 2 ], [ 5, 1 ], [ 5, 3 ] ]
gap> labels := [10,11,12,13,9];;
gap> ng := NautyDiGraphWithNodeLabels([[10,11],[10,12],[10,13],[10,9]], labels);
<A directed Nauty graph on 5 vertices>
gap> EdgesOfNautyGraph(ng);
[ [ 10, 11 ], [ 10, 12 ], [ 10, 13 ], [ 10, 9 ] ]
gap> EdgesOfNautyGraph(UnderlyingNautyGraph(ng));
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ] ]
gap> VerticesOfNautyGraph(ng);
[ 9, 10, 11, 12, 13 ]
gap> VerticesOfNautyGraph(UnderlyingNautyGraph(ng));
[ 1 .. 5 ]
```

2.1.3 NautyGraphWithNodeLabels (for IsList, IsList)

- ▷ `NautyGraphWithNodeLabels(edges, labeling)` (operation)
- ▷ `NautyDiGraphWithNodeLabels(edges, labeling)` (operation)
- ▷ `NautyColoredGraphWithNodeLabels(edges, colours, labeling)` (operation)
- ▷ `NautyColoredDiGraphWithNodeLabels(edges, colours, labeling)` (operation)

Returns: a `NautyGraph`

Construct a nauty (di)graph with node labels and optional vertex colouring, which is an object that has an underlying nauty graph. Suppose we have a graph given by a list *edges* of edges, where each edge is a list of two positive integers.

Arguments:

- *edges*: dense list of edges, encoded as pairs of positive integers. Let N denote the set of all (not necessarily consecutive) positive integers occurring in the entries of *edges*.
- *labels*: dense list of positive integers which is a map *label* from $[1 \dots |N|]$ to M .
- *colouring* (optional): dense list of colours (positive integers), indexed by the nodes of the underlying nauty graph, that is *colouring* is a map from $[1 \dots |N|]$ to a set of node colours.

This function constructs a nauty graph Γ on the nodes $\{1, \dots, |N|\}$, such that $e = [i, j]$ is an edge of Γ if and only if $[label[i], label[j]]$ is in the input list *edges*.

This function is useful, for example, if we are given a graph on a set of nodes N which is not equal to the set $\{1, \dots, |N|\}$ and we would like to translate the nodes and edges of the graph to the node set $\{1, \dots, |N|\}$ to obtain a more compact description of the graph.

Example

```
gap> ng := NautyDiGraphWithNodeLabels( [[1,8],[1,12],[1,7],[1,5]],
    [7,12,5,1,8]);
<A directed Nauty graph on 5 vertices>
gap> EdgesOfNautyGraph(ng);
[ [ 1, 8 ], [ 1, 12 ], [ 1, 7 ], [ 1, 5 ] ]
gap> ung := UnderlyingNautyGraph(ng);
<A directed Nauty graph on 5 vertices>
gap> EdgesOfNautyGraph(ung);
[ [ 4, 5 ], [ 4, 2 ], [ 4, 1 ], [ 4, 3 ] ]
```

```
DeclareSynonym("NautyColouredGraphWithNodeLabels",NautyColouredGraphWithNodeLabels);
DeclareSynonym("NautyColouredDiGraphWithNodeLabels",NautyColouredDiGraphWithNodeLabels);
```

2.1.4 NautyGraphNodeLabels (for IsNautyGraph)

▷ NautyGraphNodeLabels(*arg*)

(operation)

Returns: a List

Returns the list of node labels for a nauty (di)graph with node labels or fail else.

Nauty graphs with node labels are useful, for example, if we are given a graph on a set of nodes N which is not equal to the set $\{1, \dots, |N|\}$ and we would like to translate the nodes and edges of the graph to the node set $\{1, \dots, |N|\}$ to obtain a more compact description of the graph.

Example

```
gap> ng := NautyDiGraphWithNodeLabels( [[1,8],[1,12],[1,7],[1,5]],
    [7,12,5,1,8]);
<A directed Nauty graph on 5 vertices>
gap> EdgesOfNautyGraph(ng);
[ [ 1, 8 ], [ 1, 12 ], [ 1, 7 ], [ 1, 5 ] ]
gap> NautyGraphNodeLabels(ng);
[ 7, 12, 5, 1, 8 ]
```

Chapter 3

Nauty Traces Interface

3.1 Data Structures

3.1.1 NautyGraphFromEdges

▷ `NautyGraphFromEdges(edges)` (function)

Returns: a list of two lists

This function takes as input a list *edges* whose entries are lists of length 2, consisting of the two (possibly equal) vertices of the edges. It returns two lists, the first is a list of vertices (source vertices) where an edge originates and the second a list of corresponding vertices (range vertices) where an edge terminates. Note that when the graph is undirected, the source vertex will always be less than or equal to the range.

Example

```
gap> ng := NautyGraph( [ [1,2], [2,3], [3,4], [4,1], [3,2] ] );
<A Nauty graph with on 4 vertices>
gap> NautyGraphFromEdges( EdgesOfNautyGraph(ng));
[ [ 1, 1, 2, 3 ], [ 2, 4, 3, 4 ] ]
gap> ng := NautyDiGraph( [ [1,2], [2,3], [3,4], [4,1], [3,2] ] );
<A directed Nauty graph on 4 vertices>
gap> NautyGraphFromEdges( EdgesOfNautyGraph(ng));
[ [ 1, 2, 3, 3, 4 ], [ 2, 3, 2, 4, 1 ] ]
```

3.1.2 NautyColorData

▷ `NautyColorData(list)` (function)

Returns: two lists

This function takes as input a list of colours, which are non-negative integers. The list is interpreted as a map from the nodes of a graph to the colour of the node. This function returns two lists, called *node_list* and *color_list*. The list *node_list* is a permutation of the nodes sorted by the colour of the node as specified in the input. The second list *color_list* contains only 0, 1. The two lists together encode the colour partition of the nodes, namely the list *color_list* contains a 0 in position *i*, if *node_list*[*i*] and *node_list*[*i*+1] have different colours. Thus, if the first entry 0 in *color_list* occurs in position *j*, then the nodes stored in *node_list*[*k*] for $1 \leq k \leq j$ all have colour *list*[1].

Index

AutomorphismGroup
 for IsNautyGraph, [5](#)

CanonicalForm
 for IsNautyGraph, [6](#)

CanonicalLabeling
 for IsNautyGraph, [6](#)

CanonicalLabelingInverse
 for IsNautyGraph, [6](#)

Edges
 for IsNautyGraph, [5](#)

EdgesOfNautyGraph
 for IsNautyGraph, [5](#)

IsIsomorphicGraphs
 for IsNautyGraph, IsNautyGraph, [7](#)

IsomorphicGraphs
 for IsNautyGraph, IsNautyGraph, [7](#)

IsomorphismGraphs
 for IsNautyGraph, IsNautyGraph, [7](#)

NautyColorData, [12](#)

NautyColoredDiGraph
 for IsList, IsList, [4](#)

NautyColoredDiGraphWithNodeLabels
 for IsList, IsList, IsList, [10](#)

NautyColoredGraph
 for IsList, IsList, [4](#)

NautyColoredGraphWithNodeLabels
 for IsList, IsList, IsList, [10](#)

NautyDiGraph
 for IsList, [3](#)
 for IsList, IsInt, [3](#)

NautyDiGraphWithNodeLabels
 for IsList, IsList, [10](#)

NautyEdgeColoredDiGraph
 for IsList, [4](#)
 for IsList, IsInt, [4](#)

NautyEdgeColoredGraph
 for IsList, [4](#)
 for IsList, IsInt, [4](#)

NautyGraph
 for IsList, [3](#)
 for IsList, IsInt, [3](#)

NautyGraphFromEdges, [12](#)

NautyGraphNodeLabels
 for IsNautyGraph, [11](#)

NautyGraphWithNodeLabels
 for IsList, IsList, [10](#)

NodeLabeling
 for IsNautyGraphWithNodeLabels, [9](#)

UnderlyingNautyGraph
 for IsNautyGraphWithNodeLabels, [9](#)

VertexColoursOfNautyGraph
 for IsNautyGraph, [6](#)

VerticesOfNautyGraph
 for IsNautyGraph, [6](#)