

Handbook of Magma Functions

Part XX: Combinatorics

Version 2.30-1

Sydney

November 10, 2026

Editors

John Cannon *Wieb Bosma* *Claus Fieker* *Allan Steel*

Handbook Contributors

Eran Assaf *Geoff Bailey* *Wieb Bosma* *Gavin Brown* *Nils Bruin* *John Cannon* *Jon Carlson* *Scott Contini* *Edgar Costa* *Bruce Cox* *Brendan Creutz* *Steve Donnelly* *Tim Dokchitser* *Willem de Graaf* *Andreas-Stephan Elsenhans* *Claus Fieker* *Damien Fisher* *Volker Gebhardt* *Sergei Haller* *Michael Harrison* *Florian Hess* *Derek Holt* *David Howden* *Al Kasprzyk* *Markus Kirschmer* *David Kohel* *Axel Kohnert* *Dimitri Leemans* *Paulette Lieby* *Graham Matthews* *Scott Murray* *Eamonn O'Brien* *Dan Roozemond* *Ben Smith* *Bernd Souvignier* *William Stein* *Allan Steel* *Damien Stehlé* *Nicole Sutherland* *Don Taylor* *Bill Unger* *Alexa van der Waall* *Paul van Wamelen* *Helena Verrill* *John Voight* *Mark Watkins* *Greg White*

Production Editors

Wieb Bosma *Claus Fieker* *Allan Steel* *Nicole Sutherland*

HTML Production

Claus Fieker *Allan Steel*

Copyright © 1994–2026 The University of Sydney. All rights reserved.

Contents

157 Enumerative Combinatorics **1**
157.1 Introduction . . . 1
157.2 Combinatorial Functions . . . 1
157.3 Subsets of a Finite Set . . . 2

158 Partitions, Words, and Young Tableaux **4**
158.1 Introduction . . . 4
158.2 Partitions . . . 4
158.3 Words . . . 6
158.4 Tableaux . . . 11
158.5 Bibliography . . . 27

159 Symmetric Functions **28**
159.1 Introduction . . . 28
159.2 Creation . . . 30
159.3 Structure Operations . . . 34
159.4 Element Operations . . . 35
159.5 Transition Matrices . . . 41
159.6 Bibliography . . . 46

160 Incidence Structures and Designs **47**
160.1 Introduction . . . 47
160.2 Construction of Incidence Structures and Designs . . . 48
160.3 The Point-Set and Block-Set of an Incidence Structure . . . 52
160.4 General Design Constructions . . . 53
160.5 Elementary Invariants of an Incidence Structure . . . 57
160.6 Elementary Invariants of a Design . . . 58
160.7 Operations on Points and Blocks . . . 59
160.8 Elementary Properties of Incidence Structures and Designs . . . 61
160.9 Resolutions, Parallelisms, and Parallel Classes . . . 62
160.10 Conversion Functions . . . 65
160.11 Identity and Isomorphism . . . 66
160.12 The Automorphism Group of an Incidence Structure . . . 66
160.13 Incidence Structures, Graphs, and Codes . . . 70
160.14 Automorphisms of Matrices . . . 71
160.15 Bibliography . . . 72

161 Hadamard Matrices **73**
161.1 Introduction . . . 73
161.2 Equivalence Testing . . . 73
161.3 Associated 3-Designs . . . 75
161.4 Automorphism Group . . . 75
161.5 Databases . . . 76

162 Graphs **80**
162.1 Introduction . . . 80
162.2 Construction of Graphs and Digraphs . . . 81

162.3 Graphs with a Sparse Representation . . . 89
162.4 The Vertex–Set and Edge–Set of a Graph . . . 90
162.5 Labelled, Capacitated, and Weighted Graphs . . . 93
162.6 Standard Constructions for Graphs . . . 94
162.7 Unions and Products of Graphs . . . 99
162.8 Converting between Graphs and Digraphs . . . 100
162.9 Construction from Groups, Codes, and Designs . . . 100
162.10 Elementary Invariants of a Graph . . . 102
162.11 Elementary Graph Predicates . . . 103
162.12 Adjacency and Degree . . . 105
162.13 Connectedness . . . 106
162.14 Distances, Paths, and Circuits in a Graph . . . 112
162.15 Maximum Flow, Minimum Cut, and Shortest Paths . . . 113
162.16 Matrices and Vector Spaces Associated with a Graph or Digraph . . . 113
162.17 Spanning Trees of a Graph or Digraph . . . 113
162.18 Directed Trees . . . 114
162.19 Colourings . . . 114
162.20 Cliques, Independent Sets . . . 115
162.21 Planar Graphs . . . 119
162.22 Automorphism Group of a Graph or Digraph . . . 122
162.23 Symmetry and Regularity Properties of Graphs . . . 131
162.24 Graph Databases and Graph Generation . . . 133
162.25 Bibliography . . . 139

163 Multigraphs **140**
163.1 Introduction . . . 140
163.2 Construction of Multigraphs . . . 140
163.3 The Vertex–Set and Edge–Set of Multigraphs . . . 143
163.4 Vertex and Edge Decorations . . . 146
163.5 Standard Construction for Multigraphs . . . 151
163.6 Conversion Functions . . . 159
163.7 Elementary Invariants and Predicates for Multigraphs . . . 161
163.8 Adjacency and Degree . . . 162
163.9 Connectedness . . . 164
163.10 Spanning Trees . . . 167
163.11 Planar Graphs . . . 167
163.12 Distances, Shortest Paths, and Minimum Weight Trees . . . 170
163.13 Bibliography . . . 174

164 Networks **175**
164.1 Introduction . . . 175
164.2 Construction of Networks . . . 175
164.3 Standard Construction for Networks . . . 178
164.4 Maximum Flow and Minimum Cut . . . 183
164.5 Bibliography . . . 189

References **190**

Index **192**

Chapter 157

Enumerative Combinatorics

157.1 Introduction

This chapter presents some of the tools provided by Magma for enumerative combinatorics.

157.2 Combinatorial Functions

`Factorial(n): RngIntElt -> RngIntElt`

The factorial $n!$ for non-negative small integer n.

`NumberOfPermutations(n, k): RngIntElt, RngIntElt -> RngIntElt`

The number of permutations of n distinct objects taken k at a time.

`Binomial(n, r): RngIntElt, RngIntElt -> RngIntElt`

The binomial coefficient $\binom{n}{r}$.

`Multinomial(n, [r1, ... rn]): RngIntElt, [RngIntElt] -> RngIntElt`

Given a sequence $Q = [r_1, \ldots, r_k]$ of positive integers such that $n = r_1 + ... + r_k$, return the multinomial coefficient $\binom{n}{r_1,\ldots,r_k}$.

`Fibonacci(n): RngIntElt -> RngIntElt`

Given an integer n, this function returns the n-th Fibonacci number F_n, which can be defined via the recursion $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ for all integers n. Note that n is allowed to be negative, and that $F_{-n} = (-1)^{n+1}F_n$.

`Catalan(n): RngIntElt -> RngIntElt`

Given a small non-negative integer n, this function returns the n-th Catalan number C_n, defined via the recursion $C_0 = 1$, $C_{n+1} = C_n \cdot (4n + 2)/(n + 2)$.

`Lucas(n): RngIntElt -> RngIntElt`

Given an integer n, this function returns the n-th Lucas number L_n, which can be defined via the recursion $L_0 = 2$, $L_1 = 1$, $L_n = L_{n-1} + L_{n-2}$ for all integers n. Note that n is allowed to be negative, and that $L_{-n} = (-1)^n L_n$.

`GeneralizedFibonacciNumber(g0, g1, n): RngIntElt, RngIntElt, RngIntElt -> RngIntElt`

The nth member of the generalized Fibonacci sequence defined by $G_0 = g_0$, $G_1 = g_1$, $G_n = G_{n-1} + G_{n-2}$ for all integers n. Note that n is allowed to be negative. The Fibonacci and Lucas numbers are special cases where $(g_0, g_1) = (0, 1)$ or $(2, 1)$ respectively.

`StirlingFirst(n, k): RngIntElt, RngIntElt -> RngIntElt`

The Stirling number of the first type, $\left[{n \atop k}\right]$, where n and k are non-negative integers.

`StirlingSecond(n, k): RngIntElt, RngIntElt -> RngIntElt`

The Stirling number of the second type, $\{ {n \atop k} \}$, where n and k are non-negative integers.

`Bell(n): RngIntElt -> RngIntElt`

The nth Bell number, giving the number of partitions of a set of size n. (Not to be confused with `NumberOfPartitions(n)`, which gives the number of partitions of the *integer* n.) This is equal to the sum of `StirlingSecond(n,k)` for k between 0 and n (inclusive).

`EulerianNumber(n, r): RngIntElt, RngIntElt -> RngIntElt`

The number $E(n, r)$ of permutations p of $\{1, \ldots, n\}$ having exactly r ascents (i.e., places where $p_i < p_{i+1}$)

`HarmonicNumber(n): RngIntElt -> FldRatElt`

The nth harmonic number $H_n = \Sigma_{i=1}^{n} \frac{1}{i}$.

`BernoulliNumber(n): RngIntElt -> FldRatElt`

Returns the nth Bernoulli number B_n as a rational number.

`BernoulliApproximation(n): RngIntElt -> FldPrElt`

Returns a real approximation to the nth Bernoulli number B_n.

`BernoulliPolynomial(n): RngIntElt -> RngUPolElt`

The nth Bernoulli polynomial $B_n(x) = \sum_{k=0}^{n} \binom{n}{k} B_k x^{n-k}$ where B_n is the nth Bernoulli number.

157.3 Subsets of a Finite Set

`Subsets(S): SetEnum -> SetEnum`

The set of all subsets of the set S.

`Subsets(S, k): SetEnum, RngIntElt -> SetEnum`

The set of subsets of the set S of size k. If k is larger than the cardinality of S then the result will be empty.

`Multisets(S, k): SetEnum, RngIntElt -> SetEnum`

The set of multisets consisting of k not necessarily distinct elements of the set S.

`Subsequences(S, k): SetEnum, RngIntElt -> SetEnum`

The set of sequences of length k with elements from the set S.

`Permutations(S): SetEnum -> SetEnum;`

The set of permutations (stored as sequences) of the elements of the set S.

`Permutations(S, k): SetEnum, RngIntElt -> SetEnum;`

The set of permutations (stored as sequences) of each of the subsets of the set S of cardinality k.

Example: Petersen Graph as an Odd Graph `(ex-676132)`

The use of Subsets is illustrated in the construction of the Petersen graph as the third Odd Graph. The nth Odd Graph has its vertices in correspondence with the $(n-1)$-element subsets of $\{1 \ldots 2n-1\}$, and an edge between two vertices if and only if their corresponding sets have empty intersection.

```
> V := Subsets( {1 .. 2*n-1}, n-1) where n is 3;
> V;
{
    { 1, 5 },
    { 2, 5 },
    { 1, 3 },
    { 1, 4 },
    { 2, 4 },
    { 3, 5 },
```

(continues on next page)

(continued from previous page)

```
    { 2, 3 },
    { 1, 2 },
    { 3, 4 },
    { 4, 5 }
}
> E := { {u, v} : u,v in V | IsDisjoint(u, v) };
> Petersen := Graph< V | E >;
```

Chapter 158

Partitions, Words, and Young Tableaux

158.1 Introduction

This chapter presents some of the tools provided by Magma for partitions, words, and Young tableaux.

158.2 Partitions

A *partition* of a positive integer n is a decreasing sequence $[n_1, n_2, \ldots, n_k]$ of positive integers such that $\sum n_i = n$. The n_i are called the *parts* of the partition.

All partition functions in Magma operate only on small, positive integers.

`NumberOfPartitions(n): RngIntElt -> RngIntElt`

Given a positive integer n, return the total number of partitions of n.

`Partitions(n): RngIntElt -> [ [ RngIntElt ] ]`

Given a positive integer n, return the sequence of all partitions of n.

`Partitions(n, k): RngIntElt, RngIntElt -> [ [ RngIntElt ] ]`

Given positive integers n and k, return the sequence of all the partitions of n into k parts.

`RestrictedPartitions(n, M): RngIntElt, SetEnum -> [ [ RngIntElt ] ]`

Given a positive integer n and a set of positive integers M, return the sequence of all partitions of n, where the parts are restricted to being elements of the set M.

`RestrictedPartitions(n, k, M): RngIntElt, RngIntElt, SetEnum -> [ [ RngIntElt ] ]`

Given positive integers n and k, and a set of positive integers M, return the sequence of all partitions of n into k parts, where the parts are restricted to being elements of the set M.

`WeightedPartitions(n, w): RngIntElt, [ RngIntElt ] -> [ [ RngIntElt ] ]`

Given a positive integer n and a sequence of positive integers w, return the sequence of all weighted partitions of n with weights w, that is all sequences $a_1, \ldots, a_m$ such that $\sum_i w_i a_i = n$.

`WeightedRestrictedPartitions(n, S, w): RngIntElt, [ { RngIntElt }], [ RngIntElt ] -> [ [ RngIntElt ] ]`

Given a positive integer n, a sequence of sets of non-negative integers S, and a sequence of positive integers w, return the sequence of all weighted partitions of n with weights w, where each part is restricted to being an elements of the corresponding set in S, that is all sequences $a_1, \ldots, a_m$ such that $\sum_i w_i a_i = n$ and $a_i \in S_i$ for all $1 \leq i \leq m$.

`IsPartition(S): SeqEnum -> BoolElt`

A sequence S is considered to be a partition if it consists of weakly decreasing positive integers. A sequence is allowed to have trailing zeros, and the empty sequence is accepted as a partition (of zero).

`RandomPartition(n): RngIntElt -> SeqEnum`

Returns a weakly decreasing sequence of positive integers which is random partition of the positive integer n.

`Weight(P): SeqEnum -> RngIntElt`

Given a sequence of positive integers P which is a partition, return a positive integer which is the sum of it's parts.

`IndexOfPartition(P): SeqEnum -> RngIntElt`

Given a sequence of positive integers P which is a partition, return its lexicographical order among partitions of the same weight.

Lexicographical ordering of partitions is such that for partitions P_1 and P_2, then $P_1 > P_2$ implies that P_1 is greater in the first part which differs from P_2. The first index is zero.

Example: Partitions and Conjugacy Classes `(ex-071b39)`

The conjugacy classes of the symmetric group on n elements correspond to the partitions of n. The function `PartnToElt` below converts a partition to an element of the corresponding conjugacy class.

```
> PartitionToElt := function(G, p)
>     x := [];
>     s := 0;
>     for d in p do
>         x cat:= Rotate([s+1 .. s+d], -1);
>         s +:= d;
>     end for;
>     return G!x;
> end function;
>
> ConjClasses := function(n)
>     G := Sym(n);
>     return [ PartitionToElt(G, p) : p in Partitions(n) ];
> end function;
>
> ConjClasses(5);
[
    (1, 2, 3, 4, 5),
    (1, 2, 3, 4),
    (1, 2, 3)(4, 5),
    (1, 2, 3),
    (1, 2)(3, 4),
    (1, 2),
    Id($)
]
> Classes(Sym(5));
Conjugacy Classes
-----------------
[1]     Order 1       Length 1
        Rep Id($)

[2]     Order 2       Length 10
        Rep (1, 2)
```

(continues on next page)

(continued from previous page)

```
[3]     Order 2       Length 15
        Rep (1, 2)(3, 4)

[4]     Order 3       Length 20
        Rep (1, 2, 3)

[5]     Order 4       Length 30
        Rep (1, 2, 3, 4)

[6]     Order 5       Length 24
        Rep (1, 2, 3, 4, 5)

[7]     Order 6       Length 20
        Rep (1, 2, 3)(4, 5)
```

Example: Changing Money with Restricted Partitions (ex-d58f82)

The number of ways of changing money into five, ten, twenty, and fifty cent coins can be calculated using `RestrictedPartitions`. There is also a well known solution using generating functions, which we use as a check.

```
> coins := {5, 10, 20, 50};
> T := [#RestrictedPartitions(n, coins) : n in [0 .. 100 by 5]];
> T;
[ 1, 1, 2, 2, 4, 4, 6, 6, 9, 9, 13, 13, 18, 18, 24, 24, 31, 31, 39, 39, 49 ]
> F<t> := PowerSeriesRing(RationalField(), 101);
> &*[1/(1-t^i) : i in coins];
1 + t^5 + 2*t^10 + 2*t^15 + 4*t^20 + 4*t^25 + 6*t^30 + 6*t^35 + 9*t^40 + 9*t^45
    + 13*t^50 + 13*t^55 + 18*t^60 + 18*t^65 + 24*t^70 + 24*t^75 + 31*t^80 +
    31*t^85 + 39*t^90 + 39*t^95 + 49*t^100 + O(t^101)
```

158.3 Words

In this and following sections, words are considered to be elements of some *ordered* monoid, that is a free monoid whose generators have an explicit ordering. The most important example is the monoid of words generated by the positive integers, though Magma also allows the creation of finitely generated ordered monoids. The words in this section are developed in connection with the theory of Young tableaux (see section *Tableaux*), which are defined over some ordered set of labels (generators of an ordered monoid in Magma).

The importance of the ordering on the generators is in the definition of an equivalence known as *Knuth* equivalence. Knuth equivalence is defined by the two relations

$$\begin{aligned} yzx &\sim yxz \qquad x < y \leq z \\ xzy &\sim zxy \qquad x \leq y < z \end{aligned}$$

The *plactic* monoid is the quotient of an ordered monoid with respect to Knuth equivalence. This plactic monoid is in fact isomorphic to the monoid of tableaux over the same generators. Elements of the plactic monoid are represented by a canonical representative from the Knuth equivalence class of the ordered monoid. This canonical representative is in fact the *row word* of the corresponding tableau (see section *Tableaux*).

158.3.1 Ordered Monoids

An ordered monoid is a free monoid with an ordered basis. Magma has two different types of ordered monoids. The first is the monoid of words over the positive integers, which retain their natural ordering. The second are finitely generated monoids, whose generators may be assigned names.

`OrderedMonoid(n): RngIntElt -> MonOrd`

Given a positive integer n, return the monoid with n ordered generators.

`OrderedIntegerMonoid() -> MonOrd`

Return the ordered monoid of words over the positive integers.

`Id(O): MonOrd -> MonOrdElt`

`O ! 1: MonOrd, RngInt -> MonOrdElt`

Given an ordered monoid O, return its identity element. i.e., the null word.

`O . i: MonOrd, RngIntElt -> MonOrdElt`

Given an ordered monoid O and a positive integer i, return the ith generator of O.

`O ! [w1, ..., wn]: MonOrd, [MonOrdElt] -> MonOrdElt`

Given an ordered monoid O, and a sequence of elements from O, return the word $w_1 \cdots w_n$.

`O ! [i1, ..., in]: MonOrd, [RngIntElt] -> MonOrdElt`

Given the ordered monoid O over the positive integers, and a sequence of integers, return the word $i_1 \cdots i_n$.

`w1 eq w2: MonOrdElt, MonOrdElt -> BoolElt`

Given two words w_1 and w_2 from the same ordered monoid, return `true` if they are equal.

`w1 * w2: MonOrdElt, MonOrdElt -> MonOrdElt`

Given two words w_1 and w_2 from the same ordered monoid, return their product under word concatenation.

`IsKnuthEquivalent(w1, w2): MonOrdElt, MonOrdElt -> BoolElt`

Two words w_1 and w_2 from the same monoid are Knuth equivalent if they can be transformed into one another using elementary Knuth transformations, (defined in the introduction to this section).

`w[i]: MonOrdElt, RngIntElt -> RngElt`

Given a word w from an ordered monoid, expressed as a product of generators, return the ith generator in the product.

`ElementToSequence(w): MonOrdElt -> SeqEnum`

`Eltseq(w): MonOrdElt -> SeqEnum`

Given a word w from the ordered monoid of positive integers, return w as a sequence of integers.

`Length(w): MonOrdElt -> RngIntElt`

`# w: MonOrdElt -> RngIntElt`

Given a word w from an ordered monoid, return its length.

`Content(w): MonOrdElt -> SeqEnum[RngIntElt]`

Given a word w from an ordered monoid, return a sequence of non-negative integers denoting its content. The content of a word is a sequence where the ith position denotes the number of occurrences of the ith generator in the word.

`IsReverseLatticeWord(w): MonOrdElt -> BoolElt`

A word w from an ordered monoid is said to be a reverse lattice word (or Yamanouchi word) if for any $n > 0$, the last n letters of w have a content which is a partition.

`MaximalIncreasingSequence(w): MonOrdElt -> RngIntElt`

Given a word w from an ordered monoid, return a weakly increasing subsubsequence of w of maximal length. This sequence is not necessarily unique.

`MaximalIncreasingSequences(w, k): SeqEnum, RngIntElt -> RngIntElt`

Given a word w from an ordered monoid and some positive integer k, return a sequence of k distinct increasing subsequences of the word w, such that the maximal number of entries from w is used. Empty sequences are returned if all entries from w are used. These sequences are not necessarily unique.

Example: Ordered Monoid over the Integers (ex-9cc372)

We create the ordered monoid over the positive integers and look at a few elements.

```
> O := OrderedIntegerMonoid();
> O;
The monoid of words over the positive integers
>
> w1 := O ! [3,5,2,8];
> w1;
3 5 2 8
> w2 := O ! [6,7,2,4];
> w2;
6 7 2 4
>
> w2[3];
2
> Eltseq(w1);
[ 3, 5, 2, 8 ]
>
> w1*w2;
3 5 2 8 6 7 2 4
```

Example: Finitely Generated Ordered Monoid (ex-18b21e)

We create a finitely generated ordered monoid and create an element.

```
> O<a,b,c,d> := OrderedMonoid(4);
> O;
The monoid of words over 4 generators: a b c d
>
> w := a*b*a*d*c;
> w;
a b a d c
```

Example: Longest Increasing Subsequences (ex-31d600)

We take a word of integers and look at weakly increasing subsequences of maximal length.

```
> O := OrderedIntegerMonoid();
> w := O ! [1,3,4,2,3,4,1,2,2,3,3,2];
>
> MaximalIncreasingSequence(w);
1 1 2 2 2 3
> MaximalIncreasingSequences(w,2);
[ 1 1 2 2 2 3 , 2 3 3  ]
> MaximalIncreasingSequences(w,3);
[ 1 1 2 2 2 3 , 2 3 3 , 3 4 4  ]
> MaximalIncreasingSequences(w,4);
[ 1 1 2 2 2 3 , 2 3 3 , 3 4 4 , Id(O) ]
```

158.3.2 Plactic Monoids

The *plactic monoid* of an ordered monoid is the quotient defined by Knuth equivalence. Elements of a plactic monoid are equivalence classes of words from the original ordered monoid, and are represented by a canonical representative.

`PlacticMonoid(O): MonOrd -> MonOrd`

Given an ordered monoid O, return the plactic monoid obtained by factoring O by Knuth equivalence.

`PlacticIntegerMonoid() -> MonOrd`

Return the plactic monoid obtained by factoring the ordered monoid over the positive integers by Knuth equivalence.

`OrderedMonoid(P): MonPlc -> MonOrd`

Given a plactic monoid P, return the ordered monoid on which P is based.

`Id(P): MonPlc -> MonPlcElt`

`P ! 1: MonPlc, RngIntElt -> MonPlcElt`

Given a plactic monoid P, return its identity element which is the null word.

`P . i: MonPlc, RngIntElt -> MonPlcElt`

Given a plactic monoid P and a positive integer i, return the ith generator of P.

`P ! [u1, ..., un]: MonPlc, [MonPlcElt] -> MonPlcElt`

Given a plactic monoid P, and a sequence of elements from P, return the product $u_1 \cdots u_n$.

`P ! [i1, ..., in]: MonPlc, [RngIntElt] -> MonPlcElt`

Given the plactic monoid P over the positive integers, and a sequence of integers, return the element of P corresponding to the Knuth equivalence class of $i_1 \cdots i_n$.

`P ! w: MonPlc, MonOrdElt -> MonPlcElt`

Given a plactic monoid P, and a word w from the ordered monoid that its based on, return the element of P corresponding to the Knuth equivalence class of w.

`P ! [w1, ..., wn]: MonPlc, [MonOrdElt] -> MonPlcElt`

Given a plactic monoid P, and a sequence of elements from the ordered monoid that its based on, return the element of P corresponding to the Knuth equivalence class of $w_1 \cdots w_n$.

`P ! t: MonPlc, Tbl -> MonPlcElt`

Given a plactic monoid P and a tableau t which are both associated with the same ordered monoid, return the element of P which is uniquely associated to t.

`u1 eq u2: MonPlcElt, MonPlcElt -> BoolElt`

Given two elements u_1 and u_2 from the same plactic monoid, return `true` if they are equal.

`u1 * u2: MonPlcElt, MonPlcElt -> MonPlcElt`

Given two elements u_1 and u_2 belonging to the same plactic monoid, return their product which is inherited from word concatenation in the ordered monoid.

`Length(u): MonPlcElt -> RngIntElt`

`# u: MonPlcElt -> RngIntElt`

Given a element u from a plactic monoid, return its length. The length of a word is invariant under Knuth equivalence and so is well defined for elements of the plactic monoid.

`Content(u): MonPlcElt -> SeqEnum[RngIntElt]`

Given a word u from a plactic monoid, return a sequence of non-negative integers denoting its content. The content of a word is a sequence where the ith position denotes the number of occurrences of the ith generator in the word. The content of a word is invariant under Knuth equivalence and so is well defined for elements of the plactic monoid.

Example: Ordered and Plactic Integer Monoids (ex-0e6aa3)

We create both the ordered monoid and plactic monoid over the integers and look at several elements.

```
> O := OrderedIntegerMonoid();
> P := PlacticIntegerMonoid();
>
> w1 := O ! [2,7,4,8,1,5,9];
> w1;
2 7 4 8 1 5 9
> P!w1;
7 2 8 1 4 5 9
```

First we look at a word that is Knuth equivalent to w_1,

```
> w2 := O ! [7,2,1,8,4,5,9];
> w2;
7 2 1 8 4 5 9
>
> IsKnuthEquivalent(w1,w2);
true
> (P!w1) eq (P!w2);
true
```

and then one that is not Knuth equivalent.

```
> w3 := O ! [7,1,5,8,2,9,4];
> w3;
>
7 1 5 8 2 9 4
> IsKnuthEquivalent(w1,w3);
false
> (P!w1) eq (P!w3);
false
> P!w3;
7 5 8 1 2 4 9
```

Example: Invariants under Knuth Equivalence (ex-b4e54f)

We create a finitely generated ordered monoid, its associated plactic monoid, and look at some properties which are invariant under Knuth equivalence.

```
> O<a,b,c,d,e> := OrderedMonoid(5);
> P := PlacticMonoid(O);
> P;
The plactic monoid of words over 5 generators: a b c d e
>
> w := b*c*e*e*a*d*a*d;
> w;
b c e e a d a d
> Length(w);
8
> Content(w);
[ 2, 1, 1, 2, 2 ]
>
> u := P!w;
> u;
a a b c d d e e
```

(continues on next page)

(continued from previous page)

```
> Length(u);
8
> Content(u);
[ 2, 1, 1, 2, 2 ]
```

158.4 Tableaux

The tableaux module in Magma is loosely based on the computational algebra system SYMMETRICA developed by Adalbert Kerber and Axel Kohnert [KKL92]. The main reference for the theory of this section is "Young Tableaux" by William Fulton [Ful97b].

A Young diagram, or Ferrers diagram, is a collection of boxes, or cells, arranged in left-justified rows, with a weakly decreasing number of cells in each row. Listing the number of cells in each row gives a partition (its shape) of n, where n is the total number of cells in the diagram. Conversely, each partition corresponds to a unique Young diagram. A numbering of a Young diagram is an assignment of a positive integer to each box. More generally a numbering can be done using any ordered set of labels, in Magma this means the generators of some ordered monoid, (see section *Words*).

A Young tableau, or simply tableau, is a numbering of a Young diagram which has (i) weakly increasing entries across each row, and (ii) strictly increasing entries down each column. These tableaux form a monoid with respect to a multiplication which may be defined in either of two ways, Schensted's "bumping" row insertion, or Schützenberger's "sliding" *jeu de taquin*, (see [Ful97b] for full details).

A tableau monoid is constructed from an ordered monoid (see section *Words*), the generators of the ordered monoid comprising the tableau labels. A tableau monoid and a plactic monoid derived from the same ordered monoid are in fact isomorphic to one another.

Flipping a diagram over its main diagonal gives the conjugate diagram, its shape being the `ConjugatePartition` of the original shape.

A skew diagram or skew shape is the diagram obtained by deleting a smaller Young diagram from inside a larger one. A skew tableau is a numbering on a skew diagram obeying the same restrictions on its entries.

158.4.1 Tableau Monoids

Let O be an ordered monoid, elements of O are expressed as products of its generators. The generators of O can be used as the set of labels for a family of tableaux M, called a *tableau monoid*.

A tableau monoid has only one defining characteristic, the ordered monoid which specifies its labels.

The most important tableau monoid is the one defined over the set of all positive integers.

`TableauMonoid(O): MonOrd -> MonTbl`

`TableauMonoid(P): MonPlc -> MonTbl`

Given an ordered monoid O or a plactic monoid P, (in which case let O be the ordered monoid on which P is based), then return the tableau monoid of tableaux whose labels are the generators of O.

`TableauIntegerMonoid() -> MonTbl`

Return the tableau monoid over the positive integers.

`OrderedMonoid(M): MonPlc -> MonOrd;`

Given a tableau monoid M, return the ordered monoid from which M gets its labels.

Example: Tableau Monoid over the Integers `(ex-c8d8f5)`

We create the tableau monoid over the positive integers. We take a word in the associated ordered monoid and look at its image in the tableau monoid.

```
> M := TableauIntegerMonoid();
> M;
Monoid of Tableaux labelled by the generators of:
The monoid of words over the positive integers
>
> w := OrderedMonoid(M) ! [3,6,2,8,3,9,1];
> w;
3 6 2 8 3 9 1
> M ! w;
Tableau of shape: 4 2 1
1 3 8 9
2 6
3
```

Example: Tableau Monoid over Finitely Many Labels (ex-a88070)

We create a tableau monoid over a finite set of labels, and look at the image of a word.

```
> O<x,y,z> := OrderedMonoid(3);
> M := TableauMonoid(O);
> M;
Monoid of Tableaux labelled by the generators of:
The monoid of words over 3 generators: x y z
>
> M ! (z*y*x*z*y*y*x*y*x*z);
Tableau of shape: 5 3 2
x x x y z
y y y
z z
```

158.4.2 Creation of Tableaux

Tableaux are created from the words of some ordered monoid. For tableaux over the positive integers, the rows can be input directly as sequences. Words can be used to exactly specify the rows of a tableau, or an entire word can be mapped to a tableau using row insertion.

`Tableau(Q): SeqEnum[SeqEnum[RngIntElt]] -> Tbl`

Given a sequence of sequences of non-negative integers Q, return the tableau with the elements of Q as its rows. Zeroes will indicate skew entries. The resulting tableau must have weakly increasing rows and strictly increasing columns.

`Tableau(Q): SeqEnum[MonOrdElt] -> Tbl`

Given a sequence Q of words from some ordered monoid, return the (non-skew) tableau with the words of Q as its rows. The resulting tableau must have weakly increasing rows and strictly increasing columns.

`Tableau(S, Q): SeqEnum[RngIntElt], SeqEnum[SeqEnum[RngIntElt]] -> Tbl`

Given a sequence of nonnegative integers S which is a partition, and a sequence of sequences of positive integers Q. Create a skew tableau with skew (inner) shape given by the partition S, and the non-skew part of each row being given by the entries of Q. The resulting tableau must have weakly increasing rows and strictly increasing columns.

`Tableau(S, Q): SeqEnum[RngIntElt], SeqEnum[MonOrdElt] -> Tbl`

Given a sequence of nonnegative integers S which is a partition, and a sequence Q of words from some ordered monoid. Create a skew tableau with skew (inner) shape given by the partition S, and the non-skew part of each row being given by the words of Q. The resulting tableau must have weakly increasing rows and strictly increasing columns.

`WordToTableau(w): MonOrdElt -> Tbl`

`WordToTableau(u): MonPlcElt -> Tbl`

Given a word w from an ordered monoid, return the tableau obtained by row inserting the entries of w (from the left) into the empty tableau.

If given an element u of a plactic monoid, any word of its Knuth equivalence class is used. This map is invariant under Knuth equivalence and so is well defined for elements of the plactic monoid.

It is a surjective homomorphism from the ordered monoid to the tableau monoid, and an isomorphism from the plactic monoid to the tableau monoid.

`M ! w: MonTbl, MonOrdElt -> Tbl`

Given a tableau monoid M and a word w from its associated ordered monoid, return the tableau corresponding to w through row insertion.

`M ! u: MonTbl, MonPlcElt -> Tbl`

Given a tableau monoid M and an element u from a plactic monoid associated with the same ordered monoid as M, return the tableau corresponding to u through row insertion.

`M ! [i1, ..., in]: MonPlc, [MonPlcElt] -> MonPlcElt`

Given the tableau monoid M over the positive integers, and a sequence of positive integers, return the tableau corresponding to the word $i_1 * \cdots * i_n$.

`M ! Q: MonPlc, SeqEnum -> MonPlcElt`

Given the tableau monoid M over the positive integers, and a sequence of sequences of non-negative integers Q, return the tableau with the elements of Q as its rows. Zeroes will indicate skew entries. The resulting tableau must have weakly increasing rows and strictly increasing columns.

`M ! Q: MonPlc, SeqEnum -> MonPlcElt`

Given a tableau monoid M, and a sequence Q of words from its associated ordered monoid, return the (non-skew) tableau with the words of Q as its rows. The resulting tableau must have weakly increasing rows and strictly increasing columns.

Example: Creating Tableaux over the Integers (ex-2aa4e3)

We create a tableau over the positive integers in two ways. First we input its rows explicitly, then we use a single word which specifies the tableau. We also create a skew tableau by using zero entries in a sequence of sequences.

```
> O := OrderedIntegerMonoid();
>
> T := Tableau( [ [2,5,5,6], [5,7,9,9], [6,9] ]);
> T;
Tableau of shape: 4 4 2
2 5 5 6
5 7 9 9
6 9
> WordToTableau( O ! [6,9,5,7,9,9,2,5,5,6] );
Tableau of shape: 4 4 2
2 5 5 6
5 7 9 9
6 9
> Tableau([ [0,0,0,3,4,5], [0,1,2,4], [1,6] ]);
Skew tableau of shape: 6 4 2 / 3 1
0 0 0 3 4 5
0 1 2 4
1 6
```

Example: Skew Tableau over Finitely Many Labels (ex-53a2a5)

We create a skew tableau in a tableau monoid with finitely many labels.

```
> O<a,b,c,d> := OrderedMonoid(4);
> T := Tableau( [3,2] , [ a*a*b*d, b*d*d, a*b]);
> T;
Skew tableau of shape: 7 5 2 / 3 2
0 0 0 a a b d
0 0 b d d
a b
```

Example: Maps between Ordered, Plactic, and Tableau Monoids (ex-4125d0)

We create the ordered, plactic, and tableau monoids each over the positive integers, and examine the natural maps between them.

```
> O := OrderedIntegerMonoid();
> P := PlacticIntegerMonoid();
> M := TableauIntegerMonoid();
> w1 := O ! [4,6,2,6,9,6,2,2,1,7];
> w2 := O ! [9,4,2,1,6,2,6,2,6,7];
> w1 eq w2;
false
> IsKnuthEquivalent(w1,w2);
true
```

Knuth equivalence implies equality for the image in both the plactic and tableau monoids.

```
> (P!w1) eq (P!w2);
true
> P!w1;
9 4 2 6 6 1 2 2 6 7
>
> (M!w1) eq (M!w2);
true
> M!w1;
Tableau of shape: 5 3 1 1
1 2 2 6 7
2 6 6
4
9
```

The plactic and tableau monoids are isomorphic. We confirm that the natural image of an equivalence class from the plactic monoid is the same as the natural images of its member words.

```
> M!w1 eq M ! (P!w1);
true
```

158.4.3 Enumeration of Tableaux

Multiple tableaux may be created corresponding to specified parameters.

`StandardTableaux(P): SeqEnum[RngIntElt] -> SetEnum`

`StandardTableaux(M, P): MonTbl, SeqEnum[RngIntElt] -> SetEnum`

Returns the set of all standard tableau from the tableau monoid M of shape P, which is a partition.

If a tableau monoid M is not specified then the monoid over the positive integers is used. If M is specified then it must contain enough labels to fill the shape P (i.e., more than `Weight(P)`).

`StandardTableauxOfWeight(n): RngIntElt -> SetEnum`

StandardTableauxOfWeight(M, n): MonTbl, RngIntElt -> SetEnum

Returns the set of all standard tableau from the tableau monoid M and of weight n, which is a positive integer.

If a tableau monoid M is not specified then the monoid over the positive integers is used. If M is specified then it must contain at least n labels.

TableauxOfShape(S, m): SeqEnum[RngIntElt], RngIntElt -> SetEnum

TableauxOfShape(M, S, m): MonTbl, SeqEnum[RngIntElt], RngIntElt -> SetEnum

Given a tableau monoid M, a sequence of positive integers S which is a partition, and a positive integer m, then return the set of all tableau of shape S which use only the first m labels of the tableau monoid M.

If a tableau monoid M is not specified then the monoid over the positive integers is used. If M is specified then it must contain at least m labels.

TableauxOnShapeWithContent(S, C): SeqEnum[RngIntElt], SeqEnum[RngIntElt] -> SetEnum

TableauxOnShapeWithContent(M, S, C): MonTbl, SeqEnum[RngIntElt], SeqEnum[RngIntElt] -> SetEnum

Given a tableau monoid M, a sequence of positive integers S which is a partition, and a sequence of non-negative integers C, then return the set of all tableau from M having shape S and *content* C (see section *Properties* for definition of content). If C prescribes fewer cells than would completely fill the shape S, then the tableau within the given shape is returned.

If a tableau monoid M is not specified then the monoid over the positive integers is used. If M is specified then it must have at least as many labels as C has entries.

TableauxWithContent(C): SeqEnum[RngIntElt] -> SetEnum

TableauxWithContent(M, C): MonTbl, SeqEnum[RngIntElt] -> SetEnum

Given a tableau monoid M, and a sequence of non-negative integers C, return the set of all tableau from M with *content* C (see section *Properties* for definition of content).

If a tableau monoid M is not specified then the monoid over the positive integers is used. If M is specified then it must have at least as many labels as C has entries.

Example: Standard Tableaux of a Given Shape (ex-85d504)

We create all tableaux of a given shape, then check that the number of tableaux created agrees with the combinatorical result.

```
> P := [ 3, 2];
> S := StandardTableaux( P );
> S;
{ Tableau of shape: 3 2
 1 2 3
 4 5, Tableau of shape: 3 2
 1 3 4
 2 5, Tableau of shape: 3 2
 1 3 5
 2 4, Tableau of shape: 3 2
 1 2 5
 3 4, Tableau of shape: 3 2
 1 2 4
 3 5 }
>
> #S eq NumberOfStandardTableaux( P );
true
```

Example: Tableaux with a Given Shape and Content (ex-6d01fb)

We create all tableau over the ordered generators a, b, c, d which have shape $[4, 2]$ and exactly two a's, one b, two c's, and one d. There is in fact four of them.

```
> O<a,b,c,d> := OrderedMonoid(4);
> M := TableauMonoid(O);
> TableauxOnShapeWithContent( M, [4,2], [2,1,2,1]);
{ Tableau of shape: 4 2
a a c c
b d , Tableau of shape: 4 2
a a b c
c d , Tableau of shape: 4 2
a a b d
c c , Tableau of shape: 4 2
a a c d
b c  }
```

158.4.4 Random Tableaux

The functions in this section are based on ideas from [GNW84].

`RandomHookWalk(P, i, j): SeqEnum[RngIntElt], RngIntElt, RngIntElt -> RngIntElt, RngIntElt`

`RandomHookWalk(t, i, j): Tbl, RngIntElt, RngIntElt -> RngIntElt, RngIntElt`

A random hook walk is an essential tool for the creation of random tableau. The *hook* of a cell on a Young diagram is the cells to the right and below its position. The Young diagram is specified by either the sequence of positive integers P which is a partition, or the tableau t.

Given positive integers i and j such that (i, j) lies on the shape, the walk proceeds as follows. Starting from the specified (i, j)-th cell, move to another cell chosen at random from its hook. Repeat this process until a outside corner of the Young diagram is reached, and return the two integers representing the coordinates of this final cell.

`RandomTableau(S): SeqEnum[RngIntElt] -> Tbl`

`RandomTableau(M, S): MonTbl, SeqEnum[RngIntElt] -> Tbl`

Given a tableau monoid M, and a sequence of positive integers S which is a partition, return a random standard tableau from M of shape S.

If a tableau monoid M is not specified then the monoid over the positive integers is used. If M is specified then it must contain enough labels to fill the shape S (i.e., more than `Weight(S)`).

`RandomTableau(n): RngIntElt -> Tbl`

`RandomTableau(M, n): MonTbl, RngIntElt -> Tbl`

Given a tableau monoid M, and a positive integer n, return a (not completely) random standard tableau of weight n. The probability of any specific tableau of shape S being returned is $N_S/n!$, where N_S is the number of standard tableaux of shape S. So tableaux on more populous shapes are more likely to occur.

If a tableau monoid M is not specified then the monoid over the positive integers is used. If M is specified then it must contain at least n labels.

Example: Random Tableaux `(ex-809609)`

We numerically test the random creation of tableau of a given shape, by calculating the average percentage difference in the number of each tableau created. We find it to be less than one percent.

```
> P := [4,3,2];
> Runs := 10000;
>
> S := Setseq( StandardTableaux( P ) );
> Count := [0: i in [1..#S]];
>
> for k in [1..Runs] do
```

(continues on next page)

(continued from previous page)

```
>     T := RandomTableau( P );
>     i := Index(S,T);
>     Count[i] +:= 1;
> end for;
>
> Average :=  Runs/#S;
> Diff := [RealField(2) | Abs( (Count[i] - Average)/Average )
>                       : i in [1..#Count]];
>
> AvgDiff := (&+ Diff)/#Diff;
> AvgDiff;
0.006
```

158.4.5 Basic Access Functions

Shape(t): Tbl -> SeqEnum[RngIntElt]

OuterShape(t): Tbl -> SeqEnum

The (outer) shape of a tableau t is the sequence of positive integers denoting the (decreasing) row lengths of its Young Diagram, which results in a partition.

SkewShape(t): Tbl -> SeqEnum[RngIntElt]

InnerShape(t): Tbl -> SeqEnum[RngIntElt]

The Young diagram of a skew tableau t has a smaller Young diagram inside of it deleted. The skew, or inner, shape of a skew tableau is the sequence of positive integers denoting the (decreasing) row lengths of this deleted diagram, which results in a partition.

A non-skew tableau is a special case of a skew tableau, whose skew shape is simply the null sequence.

Trailing zeroes are added to the inner shape of t to give it the same length as the outer shape of t.

PartitionCovers(P1, P2): SeqEnum, SeqEnum -> BoolElt

Given two sequences of integers, P_1 and P_2, which are partitions, return `true` if the Young diagram of P_1 covers the Young diagram of P_2.

ConjugatePartition(P): SeqEnum -> SeqEnum

Given a sequence of integers P which is a partition, return the partition corresponding to the *conjugate* Young diagram of P (see section *Tableaux* for a full description).

Weight(t): Tbl -> RngIntElt

Given a tableau t, return the positive integer which is the number of (non-skew) cells in its Young Diagram.

SkewWeight(t): Tbl -> RngIntElt

Given a tableau t, return the positive integer which is the number of skew cells in its Young Diagram.

NumberOfRows(t): Tbl -> RngIntElt

Given a tableau t, return a positive integer which is its number of rows.

NumberOfSkewRows(t): Tbl -> RngIntElt

Given a tableau t, return a positive integer which is its number of skewed rows.

Row(t, i): Tbl, RngIntElt -> MonOrdElt

Given a tableau t, return the non-skewed entries of the ith row as a word of an ordered monoid.

Rows(t): Tbl -> SeqEnum[MonOrdElt]

Given a tableau t, return a sequence of words from an ordered monoid which are the non-skewed entries of its rows.

`Column(t, j): Tbl, RngIntElt -> MonOrdElt`

Given a tableau t, return the non-skewed entries of the jth column as a word of an ordered monoid.

`Columns(t): Tbl -> SeqEnum[MonOrdElt]`

Given a tableau t, return a sequence of words from an ordered monoid which are the non-skewed entries of its columns.

`RowSkewLength(t, i): Tbl, RngIntElt -> RngIntElt`

Given a tableau t, return the length of the skewed portion of the ith row (zero if no skewed portion).

`ColumnSkewLength(t, j): Tbl, RngIntElt -> RngIntElt`

Given a tableau t, return the length of the skewed portion of the jth column (zero if no skewed portion).

`FirstIndexOfRow(t, i): Tbl, RngIntElt -> RngIntElt`

Given a tableau t and a positive integer i, return the index of the first non-skew entry of the ith row of t. If the row has no non-skew entries then the index will be one greater than the length of the row (i.e., out of bounds).

`LastIndexOfRow(t, i): Tbl, RngIntElt -> RngIntElt`

`RowLength(t, i): Tbl, RngIntElt -> RngIntElt`

Given a tableau t and a positive integer i, return the index of the last entry of the ith row of t, which is the length of the ith row.

`FirstIndexOfColumn(t, j): Tbl, RngIntElt -> RngIntElt`

Given a tableau t and a positive integer j, return the index of the first non-skew entry of the jth column of t. If the column has no non-skew entries then the index will be one greater than the length of the column (i.e., out of bounds).

`LastIndexOfColumn(t, j): Tbl, RngIntElt -> RngIntElt`

`ColumnLength(t, j): Tbl, RngIntElt -> RnfIntElt`

Given a tableau t and a positive integer j, return the index of the last entry in the jth column of t, which is the length of the jth column.

Example: Properties of a Skew Tableau (ex-ac4145)

Given a skew tableau, we access a number of its structural properties.

```
> T := Tableau( [3,3,2] , [ [4, 6 ] ,[5, 9], [1, 8] ]);
> T;
Skew tableau of shape: 5 5 4 / 3 3 2
0 0 0 4 6
0 0 0 5 9
0 0 1 8
> Shape(T);
[ 5, 5, 4 ]
> Weight(T);
6
> SkewWeight(T);
8
> NumberOfRows(T);
3
> NumberOfSkewRows(T);
3
> RowSkewLength(T, 2);
3
> Row(T, 2);
5 9
> ColumnSkewLength(T, 3);
```

(continues on next page)

(continued from previous page)

```
2
> Column(T, 3);
1
```

158.4.6 Properties

HookLength(t, i, j): Tbl, RngIntElt, RngIntElt -> RngIntElt

HookLength(P, i, j): SeqEnum[RngIntElt], RngIntElt, RngIntElt -> RngIntElt

The *hook* of a cell on a Young diagram comprises the cells to the right and below its position. The Young diagram is specified by either the sequence of positive integers P which is a partition or the tableau t. The number of cells contained in a hook is its *length*.

For positive integers i and j such that (i, j) is the co-ordinates of a cell lying on the specified Young diagram, a positive integer is returned which is the length of the hook of that cell.

Content(t): Tbl -> SeqEnum

Given a tableau t, return a sequence of non-negative integers which is its *content*.

The content of t is such that its ith value denotes the number of times the ith label occurs in t.

Word(t): Tbl -> MonOrdElt

RowWord(t): Tbl -> MonOrdElt

Given a tableau t, its row word is formed by reading its entries from left to right, bottom to top. The result is a word of an ordered monoid.

ColumnWord(t): Tbl -> SeqEnum

Given a tableau t, its column word is formed by reading its labels from bottom to top, left to right. The result is a word of an ordered monoid.

IsStandard(t): Tbl -> BoolElt

Given a tableau t of weight n, then t is standard if and only if its entries are exactly the first n labels of its parent monoid. So t is standard if and only if it has a content of the form $[1, 1, \ldots, 1]$.

IsSkew(t): Tbl -> BoolElt

Returns `true` if the tableau t is a skew tableau.

IsLittlewoodRichardson(t): Tbl -> BoolElt

Given a tableau t, then it is a Littlewood–Richardson tableau if the content of t forms a reverse lattice word, (see section *Words* for a definition of a reverse lattice word).

Example: Checking Whether a Tableau Is Standard (ex-9a2ad2)

We create a tableau which is not standard. Examining its content shows a simple to way to transform it into a standard tableau.

```
> O := OrderedIntegerMonoid();
> T := WordToTableau( O ! [8,1,7,3,6,2,5,9] );
> T;
Tableau of shape: 4 2 1 1
 1 2 5 9
 3 6
 7
 8
> Content(T);
[ 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
> IsStandard(T);
false
>
```

(continues on next page)

(continued from previous page)

```
> RowInsert(~T, 4);
> T;
Tableau of shape: 4 2 1 1 1
 1 2 4 9
 3 5
 6
 7
 8
> Content(T);
[ 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
> IsStandard(T);
true
```

Example: Knuth Equivalence of Row and Column Words (ex-61310d)

Given a tableau, check that the row and column words are Knuth equivalent.

```
> T := Tableau( [2, 2], [ [1,2,3], [4,6,6], [1,5], [2,6] ]);
> T;
Skew tableau of shape: 5 5 2 2 / 2 2
0 0 1 2 3
0 0 4 6 6
1 5
2 6
>
> RW := RowWord(T);
> RW;
2 6 1 5 4 6 6 1 2 3
> CW := ColumnWord(T);
> CW;
2 1 6 5 4 1 6 2 6 3
> IsKnuthEquivalent(RW, CW);
true
```

158.4.7 Operations

Several operations exist to manipulate tableaux and allow them to interact.

`t1 eq t2: MonPlcElt, MonPlcElt -> BoolElt`

Given two tableaux t_1 and t_2, return `true` if they are equal.

`t1 * t2: Tbl, Tbl -> Tbl`

Given two tableaux t_1 and t_2, return another tableau which is their product. The product of two tableaux can be defined using either row insertion or *jeu de taquin*.

`DiagonalSum(t1, t2): Tbl, Tbl -> Tbl`

The `DiagonalSum` of two tableau $t1$ and $t2$ is formed by first building a rectangle of empty cells, with the same number of columns as $t1$ and the same number of rows as $t2$. Then $t1$ is attached below the rectangle and $t2$ attached to right of it.

`Conjugate(t): Tbl -> Tbl`

Given a tableau t with strictly increasing rows, the rows and columns of t are transposed to form a new tableau on the conjugate Young diagram.

`JeuDeTaquin(~t, i, j): Tbl, RngIntElt, RngIntElt`

`JeuDeTaquin(t, i, j): Tbl, RngIntElt, RngIntElt -> Tbl`

Given a skew tableau t, and the co-ordinates (i, j) which must be a skew cell that is an *inside corner* (a corner on the skew shape of t). The skewed (i, j)th cell is removed using *jeu de taquin*.

`JeuDeTaquin(~t): Tbl`

`JeuDeTaquin(t): Tbl -> Tbl`

`Rectify(~t): Tbl`

`Rectify(t): Tbl -> Tbl`

Given a tableau t, remove all skewed cells successively using *jeu de taquin*.

`InverseJeuDeTaquin(~t, i, j): Tbl, RngIntElt, RngIntElt`

`InverseJeuDeTaquin(t, i, j): Tbl, RngIntElt, RngIntElt -> Tbl`

Given a tableau t, and co-ordinates (i, j) which must lie on the outside of t resulting in a new and valid shape. A new skew tableau with skew degree one higher than t is created. This is achieved by adding a new skew cell in the (i, j)th position, then performing *jeu de taquin* in reverse, which moves the new entry to create a valid skew tableau.

`RowInsert(~t, w): Tbl, MonOrdElt`

`RowInsert(t, w): Tbl, MonOrdElt -> Tbl`

`RowInsert(~t, u): Tbl, MonPlcElt`

`RowInsert(t, u): Tbl, MonPlcElt -> Tbl`

Given a tableau t and a word w from the same ordered monoid that t is associated to, insert w into t using Schensted's row insertion algorithm.

If given an element u of a plactic monoid, any word of its Knuth equivalence class is used. This map is invariant under Knuth equivalence and so is well defined for elements of the plactic monoid.

`RowInsert(~t, x): Tbl, RngIntElt`

`RowInsert(t, x): Tbl, RngIntElt -> Tbl`

Given a tableau t over the positive integers, and a positive integer x, insert x into t using the Schensted's row insertion algorithm.

`InverseRowInsert(~t, i, j): Tbl, RngIntElt, RngIntElt`

`InverseRowInsert(t, i, j): Tbl, RngIntElt, RngIntElt -> Tbl, MonOrdElt`

Given a tableau t and positive integers i and j such that (i, j) corresponds to an outside corner cell of the tableau t, then the row insertion algorithm is applied in reverse to remove the specified entry. In the functional form, both the generated tableau and the value of the removed entry as a word of an ordered monoid are returned.

Example: Jeu de Taquin (ex-00f593)

The order with which the skew entries of a skew tableau are removed using jeu de taquin is irrelevant to the final tableau produced. We give an example of this.

```
> T := Tableau([2,1], [ [2], [3,4] ,[3] ]);
> T1 := T;
> T;
Skew tableau of shape: 3 3 1 / 2 1
 0 0 2
 0 3 4
 3
>
> JeuDeTaquin(~T, 1, 2); T;
Skew tableau of shape: 3 2 1 / 1 1
 0 2 4
 0 3
 3
> JeuDeTaquin(~T, 2, 1); T;
Skew tableau of shape: 3 2 / 1
 0 2 4
 3 3
```

(continues on next page)

(continued from previous page)

```
> JeuDeTaquin(~T, 1, 1); T;
Tableau of shape: 3 1
 2 3 4
 3
>
> JeuDeTaquin(~T1, 2, 1); T1;
Skew tableau of shape: 3 3 / 2
 0 0 2
 3 3 4
> JeuDeTaquin(~T1, 1, 2); T1;
Skew tableau of shape: 3 2 / 1
 0 2 4
 3 3
> JeuDeTaquin(~T1, 1, 1); T1;
Tableau of shape: 3 1
 2 3 4
 3
>
> T eq T1;
true
```

Example: Two Ways of Multiplying Tableaux `(ex-3dbdc0)`

We manually compare the two different methods of multiplying tableau.

```
> O<a,b,c,d,e,f,g,h> := OrderedMonoid(8);
> T1 := Tableau( [ a*c*f, d*g]);
> T2 := Tableau( [ b*b*e*f, d*g*h]);
```

First using row insert,

```
> w := Word(T2);
> w;
d g h b b e f
> Res1 := RowInsert(T1, w);
> Res1;
Tableau of shape: 5 4 2 1
a b b e f
c d g h
d f
g
```

and then using JeuDeTaquin.

```
> Res2 := DiagonalSum(T1, T2);
> Res2;
Skew tableau of shape: 7 6 3 2 / 3 3
0 0 0 b b e f
0 0 0 d g h
a c f
d g
> Rectify(~Res2);
> Res2;
Tableau of shape: 5 4 2 1
a b b e f
c d g h
d f
```

(continues on next page)

(continued from previous page)

```
g
```

Now we check our results.

```
> Res1 eq Res2;
true
> Res1 eq (T1*T2);
true
```

158.4.8 The Robinson-Schensted-Knuth Correspondence

While the map from the plactic monoid to tableaux is an isomorphism, many different words from the original ordered monoid map to the same tableau. However a bijective map exists between an ordered monoid and *pairs* of tableau of the same shape, the second of which being a standard tableau. This is known as the *Robinson correspondence*.

By removing the restriction that the second tableau be standard, this correspondence is extended further to a bijective map with all matrices with non-negative integers. This is known as the Robinson-Schensted-Knuth (RSK) correspondence.

This latter correspondence can also be made to all lexicographically ordered pairs of words (both having the same length).

`LexicographicalOrdering(~w1, ~w2): MonOrdElt, MonOrdElt`

`LexicographicalOrdering(w1, w2): MonOrdElt, MonOrdElt -> MonOrdElt, MonOrdElt`

Given two words w_1 and w_2 of the same length, place them in increasing lexicographical order (the functional version returns two new words). Lexicographical ordering on the pairs $(w_1[i], w_2[i])$ is determined primarily by the comparison $w_2[i] < w_2[j]$, but if $w_2[i] = w_2[j]$ then the comparison $w_1[i] < w_1[j]$ is taken into account.

`IsLexicographicallyOrdered(w1, w2): MonOrdElt, MonOrdElt -> boolean`

Given two words w_1 and w_2, return `true` if the pairs $(w_1[i], w_2[i])$ are ordered according to the ordering described in `LexicographicalOrdering`.

`RSKCorrespondence(w): SeqEnum[RngIntElt] -> Tbl, Tbl`

Given a word w, use the bijective map described by the Robinson-Schensted-Knuth correspondence to return two tableau t_1 and t_2. The first tableau, t_1, will be labelled by the entries of w, while t_2 is a standard tableau over the positive integers.

The map described by the correspondence is as follows. Map the word w to t_1 by row inserting its entries into the empty tableau, (as is done in `WordToTableau`). As t_1 is being generated, the insertion tableau t_2 is built simultaneously. As $w[k]$ is inserted into t_1, an additional cell is added to its shape. A new cell is added to t_2 in exactly the same place, and it is labelled with the integer k.

`InverseRSKCorrespondenceSingleWord(t1, t2): Tbl, Tbl -> MonOrdElt`

Given a pair of tableaux t_1, t_2, of the same shape, where t_2 is over the positive integers, return the preimage of t_1 and t_2 using the bijective map described by the Robinson-Schensted-Knuth correspondence.

The result is returned as a single word of the ordered monoid of t_1, as described in the single word version of `RSKCorrespondence`.

`RSKCorrespondence(w1, w2): SeqEnum[RngIntElt], SeqEnum[RngIntElt] -> Tbl, Tbl`

Given two lexicographically ordered words w_1 and w_2 of the same length, where w_2 must be over the positive integers, use the bijective map described by the Robinson-Schensted-Knuth correspondence to return two tableau t_1 and t_2. The first tableau, t_1, will be labelled by the entries of w_1, while t_2 will be labelled by the entries of w_2.

The map described by the correspondence is as follows. Map the word w_1 to t_1 by row inserting its entries into the empty tableau, (as is done in `WordToTableau`). As t_1 is being generated, the

insertion tableau t_2 is built simultaneously. As $w[k]$ is inserted into t_1, an additional cell is added to its shape. A new cell is added to t_2 in exactly the same place, and it is labelled with the integer $w_2[k]$.

`InverseRSKCorrespondenceDoubleWord(t1, t2): Tbl, Tbl -> MonOrdElt, MonOrdElt`

Given a pair of tableaux t_1, t_2, of the same shape, where t_2 is over the positive integers, return the preimage of t_1 and t_2 using the bijective map described by the Robinson-Schensted-Knuth correspondence.

The result is returned as words of the ordered monoids of t_1 and t_2 respectively, as described in the double word version of `RSKCorrespondence`.

`RSKCorrespondence(M): Mtrx -> Tbl, Tbl`

Given a matrix M of non-negative integers, use the bijective map described by the Robinson-Schensted-Knuth correspondence to return two tableau t_1 and t_2 over the positive integers.

The map described by the correspondence is as follows. Initialise two words over the positive integers, w_1 and w_2, as null words. For the (i, j)th entry of M, say m_{ij}, append i to w_1 m_{ij} times, and append j to w_2 m_{ij} times. These two words are then used in the correspondence described in the above version of `RSKCorrespondence`.

`InverseRSKCorrespondenceMatrix(t1, t2): Tbl, Tbl -> Mat`

Given a pair of tableaux t_1, t_2, of the same shape, both over the positive integers, return the preimage of t_1 and t_2 using the bijective map described by the Robinson-Schensted-Knuth correspondence.

The result is a matrix of non-negative integers, as described in the matrix version of `RSKCorrespondence`.

Example: RSK Correspondence for a Single Word `(ex-195c23)`

We take two words from the ordered monoid over the positive integers. Although they correspond to the same tableau under the natural mapping into the tableau monoid, we show they have a different image under the RSK correspondence. That is, they have different *insertion* tableaux.

```
> O := OrderedIntegerMonoid();
> M := TableauMonoid(O);
> a := O ! [4,7,2,8,4,9,2];
> a;
4 7 2 8 4 9 2
> b := O ! [7,4,2,4,2,8,9];
> b;
7 4 2 4 2 8 9
```

Now we will see that a and b have the same image in the tableau monoid, but that they are distinguished under the RSK correspondence.

```
> (M!a) eq (M!b);
true
> Ta1, Ta2 := RSKCorrespondence(a);
> Tb1, Tb2 := RSKCorrespondence(b);
> Ta1 eq Tb1;
true
> Ta2 eq Tb2;
false
> Ta2;
Tableau of shape: 4 2 1
1 2 4 6
3 5
7
> Tb2;
```

(continues on next page)

(continued from previous page)

```
Tableau of shape: 4 2 1
1 4 6 7
2 5
3
```

To check that the map is indeed bijective we find a and b as the pre-images of the created tableaux.

```
> InverseRSKCorrespondenceSingleWord(Ta1,Ta2 );
4 7 2 8 4 9 2
> InverseRSKCorrespondenceSingleWord(Tb1,Tb2 );
7 4 2 4 2 8 9
```

Example: RSK Correspondence for Two Words (ex-af7728)

We apply the RSK correspondence to two arbitrary words, the first from a finitely generated ordered monoid. The second word must always be over the positive integers. The first step is to put them into lexicographical order.

```
> O1<a,b,c,d,e,f> := OrderedMonoid(6);
> O2 := OrderedIntegerMonoid();
>
> w1 := e*a*e*b*f*d*a;
> w2 := O2 ! [3,8,2,8,3,3,6];
> LexicographicalOrdering(~w1, ~w2);
> w1, w2;
e d e f a a b  2 3 3 3 6 8 8
```

Now we use the correspondence,

```
> T1, T2 := RSKCorrespondence(w1, w2);
> T1;
Tableau of shape: 3 3 1
a a b
d e f
e
> T2;
Tableau of shape: 3 3 1
2 3 3
3 8 8
6
```

and check that the inverse is correct.

```
> InverseRSKCorrespondenceDoubleWord(T1, T2);
e d e f a a b  2 3 3 3 6 8 8
```

Example: RSK Correspondence for Matrices (ex-379af5)

We give an example of the bijective map provided by the RSK correspondence between matrices of non-negative integers and pairs of tableaux of the same shape.

```
> M := Matrix(2,3,[0,0,2,3,1,2]);
> M;
[0 0 2]
[3 1 2]
> T1, T2 := RSKCorrespondence(M);
> T1;
Tableau of shape: 6 2
```

(continues on next page)

(continued from previous page)

```
1 1 1 2 3 3
3 3
> T2;
Tableau of shape: 6 2
1 1 2 2 2 2
2 2
> InverseRSKCorrespondenceMatrix(T1, T2);
[0 0 2]
[3 1 2]
```

Looking now at preimage of the tableaux in the double word format, the (i, j) co-ordinates of entries of M can clearly be seen with the correct multiplicities.

```
> wj, wi := InverseRSKCorrespondenceDoubleWord(T1, T2);
> wi;
1 1 2 2 2 2 2 2
> wj;
3 3 1 1 1 2 3 3
```

Example: RSK Correspondence and Inverse Permutations (ex-48940b)

We illustrate the remarkable property that a permutation p (in image notation) corresponds to the tableaux pair $(T1, T2)$ if and only if the inverse permutation p^{-1} corresponds to the tableaux pair $(T2, T1)$.

```
> O := OrderedIntegerMonoid();
> n := Random([1..100]);
> G := SymmetricGroup(n);
> p := Random(G);
>
> T1, T2 := RSKCorrespondence( O ! Eltseq(p) );
>
> p1 := InverseRSKCorrespondenceSingleWord( T2, T1);
> p1 := G ! Eltseq(p1);
>
> p1 eq p^-1;
true
```

158.4.9 Counting Tableaux

`NumberOfStandardTableaux(P): SeqEnum -> RngIntElt`

Given a sequence of positive integers P which is a partition, return the number of standard tableaux of shape P. This is the same as the number of Knuth equivalent words corresponding to each standard tableaux of shape P.

`NumberOfStandardTableauxOnWeight(n): RngIntElt -> RngIntElt`

Given a positive integer n, return the number of standard tableaux having weight n.

`NumberOfTableauxOnAlphabet(P, m): SeqEnum, RngIntElt -> RngIntElt`

Given a sequence of positive integers P which is a partition, return the number of tableau of shape P with entries from $[1, \ldots, m]$.

`KostkaNumber(S, C): SeqEnum[RngIntElt], SeqEnum[RngIntElt] -> RngIntElt`

Given a sequence of positive integers S forming a partition, and a sequence of non-negative integers C, return the number of tableau having shape S and content C. If C prescribes fewer cells than would completely fill the shape S, then the number of tableau within the given shape is returned.

Example: Counting Standard Tableaux (ex-d0a453)

There is a correspondence between the number of standard tableaux on a given shape, and the number of words associated with each of those tableaux. We manually count the number of words corresponding to a specific standard tableau, and show that this is equal to the number of standard tableaux on that shape.

```
> O := OrderedIntegerMonoid();
> T := Tableau( [ [ 1, 2, 3, 4, 7],
>                 [ 5, 8],
>                 [ 6]    ] );
> T;
Tableau of shape: 5 2 1
 1 2 3 4 7
 5 8
 6
>
> n := Weight(T);
> G := SymmetricGroup(n);
> S := [1..n];
>
> Count := 0;
> for p in G do
>     T1 := WordToTableau( O ! (S^p) );
>     if T1 eq T then
>         Count +:= 1;
>     end if;
> end for;
>
> Count;
64
> NumberOfStandardTableaux( Shape(T) );
64
```

Example: Counting Single-Column Tableaux (ex-3167f1)

If a tableaux is constructed of the shape $[1, 1, 1, \ldots, 1]$ then the entries must be strictly increasing, hence non-repeating. So for such a tableau of length n on an alphabet of m symbols, each subset of n symbols will correspond to one single distinct tableau. So the number of tableau will be $\binom{m}{n}$.

```
> m := Random(1,50);
> n := Random(1,m);
>
> Shape := [ 1 : i in [1..n] ];
> Binomial(m,n) eq NumberOfTableauxOnAlphabet(Shape, m);
true
```

158.5 Bibliography

- [Ful97b] William Fulton. *Young Tableaux*. Cambridge University Press, Cambridge, 1997. ISBN 0-521-56144-2; 0-521-56724-6.
- [GNW84] Curtis Greene, Albert Nijenhuis, and Herbert S. Wilf. Another probabilistic method in the theory of Young tableaux. *J. Combin. Theory Ser. A*, 37(2):127–135, 1984.
- [KKL92] Adalbert Kerber, Axel Kohnert, and Alain Lascoux. SYMMETRICA, an object oriented computer-algebra system for the symmetric group. *J. Symbolic Comp.*, 14(2-3):195–203, 1992. http://www.mathe2.uni-bayreuth.de/axel/symneu_engl.html.

Chapter 159

Symmetric Functions

159.1 Introduction

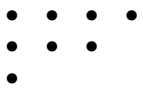
A *symmetric function* is a polynomial which is invariant under permutations of its indeterminates. The symmetric functions over a commutative ring with unity, with an arbitrary number of indeterminates, form an algebra, denoted by Λ.

The symmetric functions of fixed degree n form a submodule of Λ denoted by Λ^n. When analysing and computing with elements of Λ, there are 5 important bases to consider. These are the bases consisting of the *Schur*, *Homogeneous*, *Power Sum*, *Elementary*, and *Monomial* symmetric functions. The size of any basis of Λ^n is equal to the number of partitions of weight n, and each basis has its elements indexed by those partitions.

A *partition* is a weakly decreasing sequence of positive integers. So, for example, $[5, 3, 1]$ is a partition of weight 9, and $[3, 1]$ is a partition of weight 4. For more information on partitions see Section *Partitions*.

Magma allows computations with symmetric functions in any of the 5 bases, and any symmetric function can be expressed in any of the different bases. The main reference on the theory of symmetric functions used here is the book of Macdonald [Mac95].

In order to understand the different symmetric function bases, it is necessary to have an understanding of *Young Tableaux*. Firstly, for a given partition let us visualise its *shape* making a diagram. We arrange "boxes" in left-justified rows corresponding to the entries of the partition. So for example, the partition $[4, 3, 1]$ of weight 8 gives



This is known as a *Ferrers diagram*.

Now, to create a Young tableau, we must fill these boxes with integer entries, but subject to some restrictions. We require that each row of the tableau must be weakly increasing, and each column must be strictly increasing. So, for example, a possible tableau of shape $[4, 3, 1]$ would be

$$T = \begin{array}{cccc} 1 & 3 & 3 & 4 \\ 2 & 4 & 5 & \\ 4 & & & \end{array}$$

A tableau can be mapped onto a monomial by treating each entry in the tableau as an indeterminate. So, the above tableau will correspond to the monomial $T \mapsto x^T = x_1 x_2 x_3^2 x_4^3 x_5$

With this connection between tableaux and monomials we are now able to define the bases of an algebra of symmetric functions.

An important point to realise is that when dealing with a symmetric function, the number of indeterminates of the corresponding polynomial is neither fixed nor important. In fact a symmetric function has

an image as a symmetric polynomial in an arbitrary number of indeterminates, even an infinite number. Magma is only equipped to deal with polynomial rings with finite rank.

Since the partitions of weight n prescribe the symmetric function basis elements of degree n which generate Λ^n, (the symmetric polynomials in n indeterminates), then the number of indeterminates being used will normally be equal to the degree of the symmetric functions considered.

- Schur Functions

Given a partition $\lambda = [\lambda_1, \lambda_2, \ldots, \lambda_l]$ of weight n, the Schur function s_λ is given by $s_\lambda = \sum_T x^T$ where the sum is taken over all Young tableaux with shape λ, whose entries are all less than or equal to m, (the number of required indeterminates).

- Elementary Functions

For a positive integer k, define the k-th elementary symmetric function, e_k, as the Schur function of $[1^k]$. Then, for any partition $\lambda = [\lambda_1, \lambda_2, \ldots, \lambda_l]$, we define the elementary symmetric function indexed by λ to be $e_\lambda = \prod_i e_{\lambda_i}$.

- Homogeneous Functions

For a positive integer k, define the k-th homogeneous symmetric function, h_k, as the Schur function of $[k]$. Then, for any partition $\lambda = [\lambda_1, \lambda_2, \ldots, \lambda_l]$, we define the homogeneous symmetric function indexed by λ to be $h_\lambda = \prod_i h_{\lambda_i}$.

- Monomial Functions

For a partition $\lambda = [\lambda_1, \lambda_2, \ldots, \lambda_l]$, let x^λ be given by $x_1^{\lambda_1} x_2^{\lambda_2} \cdots x_l^{\lambda_l}$. The monomial symmetric function m_λ is the sum over the orbits of x^λ under the action of the symmetric group. Where the number of indeterminates exceeds the length of the partition, λ can be considered to have trailing zeros.

- Power Sums

For a positive integer k, define the k-th power sum symmetric function, p_k, as the monomial function of $[k]$. Then, for any partition $\lambda = [\lambda_1, \lambda_2, \ldots, \lambda_l]$, we define the power sum symmetric function indexed by λ to be $p_\lambda = \prod_i p_{\lambda_i}$.

For each of these classes of functions, the set of all functions indexed by partitions of weight n forms a basis for Λ^n. Taking functions indexed by partitions of *all* weights gives a basis of Λ.

It is possible to define an inner product on Λ by requiring that the bases m_λ and h_λ be dual to each other:

$$\langle m_\lambda, h_{\lambda'} \rangle = \delta_{\lambda,\lambda'}$$

With respect to this inner product, the Schur functions are an orthonormal basis of Λ^n and Λ and the power sum basis is an orthogonal basis.

Example: Symmetric Functions in Any Number of Variables (ex-1cbf7a)

We give an example illustrating the correspondence between symmetric functions written with respect to a symmetric function basis and symmetric polynomials. While the degree of the symmetric function fixes the degree of the resulting polynomial, the number of indeterminates is arbitrary.

```
> Q := Rationals();
> S := SFASchur(Rationals());
> e := S.[2,1];
> P2<[x]> := PolynomialRing(Q, 2);
> P2 ! e;
x[1]^2*x[2] + x[1]*x[2]^2
> P3<[y]> := PolynomialRing(Q, 3);
> P3 ! e;
y[1]^2*y[2] + y[1]^2*y[3] + y[1]*y[2]^2 + 2*y[1]*y[2]*y[3] + y[1]*y[3]^2 +
    y[2]^2*y[3] + y[2]*y[3]^2
> P4<[z]> := PolynomialRing(Q, 4);
```

(continues on next page)

(continued from previous page)

```
> P4 ! e;
z[1]^2*z[2] + z[1]^2*z[3] + z[1]^2*z[4] + z[1]*z[2]^2 + 2*z[1]*z[2]*z[3] +
    2*z[1]*z[2]*z[4] + z[1]*z[3]^2 + 2*z[1]*z[3]*z[4] + z[1]*z[4]^2 +
    z[2]^2*z[3] + z[2]^2*z[4] + z[2]*z[3]^2 + 2*z[2]*z[3]*z[4] +
    z[2]*z[4]^2 + z[3]^2*z[4] + z[3]*z[4]^2
>
> P5<[w]> := PolynomialRing(Q, 5);
> P5 ! e;
w[1]^2*w[2] + w[1]^2*w[3] + w[1]^2*w[4] + w[1]^2*w[5] + w[1]*w[2]^2 +
    2*w[1]*w[2]*w[3] + 2*w[1]*w[2]*w[4] + 2*w[1]*w[2]*w[5] + w[1]*w[3]^2 +
    2*w[1]*w[3]*w[4] + 2*w[1]*w[3]*w[5] + w[1]*w[4]^2 + 2*w[1]*w[4]*w[5] +
    w[1]*w[5]^2 + w[2]^2*w[3] + w[2]^2*w[4] + w[2]^2*w[5] + w[2]*w[3]^2 +
    2*w[2]*w[3]*w[4] + 2*w[2]*w[3]*w[5] + w[2]*w[4]^2 + 2*w[2]*w[4]*w[5] +
    w[2]*w[5]^2 + w[3]^2*w[4] + w[3]^2*w[5] + w[3]*w[4]^2 +
    2*w[3]*w[4]*w[5] + w[3]*w[5]^2 + w[4]^2*w[5] + w[4]*w[5]^2
```

159.2 Creation

159.2.1 Creation of Symmetric Function Algebras

Symmetric functions are symmetric polynomials over some coefficient ring, hence the algebra of symmetric functions is defined by specifying this ring. There are five standard bases that can be used to express symmetric functions. In Magma there are separate constructions for creating an algebra using each distinct basis.

SymmetricFunctionAlgebra(R): Rng -> AlgSym

SFA(R): Rng -> AlgSym

```
Basis: MonStgElt                    Default: "Schur"
```

Given a ring R, return the algebra of symmetric functions over R. By default this algebra will use the basis of `"Schur"` functions, though the parameter `Basis` allows the user to specify that the basis be one of the `"Homogeneous"`, `"PowerSum"`, `"Elementary"`, or `"Monomial"` functions.

SymmetricFunctionAlgebraSchur(R): Rng -> AlgSym

SFASchur(R): Rng -> AlgSym

Given a commutative ring with unity R, create the algebra of symmetric functions (polynomials) with coefficients from R. Its elements are expressed in terms of the basis of *Schur* symmetric functions, which are indexed by partitions.

SymmetricFunctionAlgebraHomogeneous(R): Rng -> AlgSym

SFAHomogeneous(R): Rng -> AlgSym

Given a commutative ring with unity R, create the algebra of symmetric functions (polynomials) with coefficients from R. Its elements are expressed in terms of the basis of *Homogeneous* symmetric functions, which are indexed by partitions.

SymmetricFunctionAlgebraPower(R): Rng -> AlgSym

SFAPower(R): Rng -> AlgSym

Given a commutative ring with unity R, create the algebra of symmetric functions (polynomials) with coefficients from R. Its elements are expressed in terms of the basis of *Power Sum* symmetric functions, which are indexed by partitions.

SymmetricFunctionAlgebraElementary(R): Rng -> AlgSym

SFAElementary(R): Rng -> AlgSym

Given a commutative ring with unity R, create the algebra of symmetric functions (polynomials) with coefficients from R. Its elements are expressed in terms of the basis of *Elementary* symmetric functions, which are indexed by partitions.

SymmetricFunctionAlgebraMonomial(R): Rng -> AlgSym

SFAMonomial(R): Rng -> AlgSym

Given a commutative ring with unity R, create the algebra of symmetric functions (polynomials) with coefficients from R. Its elements are expressed in terms of the basis of *Monomial* symmetric functions, which are indexed by partitions.

Example: Symmetric Function Algebras with Different Bases (ex-0b7c9a)

A symmetric polynomial can be expressed in terms of any of the 5 standard bases.

```
> R := Rationals();
> S := SymmetricFunctionAlgebraSchur(R);
Symmetric Algebra over Rational Field, Schur symmetric functions as basis
> E := SymmetricFunctionAlgebra(R : Basis := "Elementary");
Symmetric Algebra over Rational Field, Elementary symmetric functions as basis
> f1 := S.[2,1];
> f1;
S.[2,1]
>
> f2 := E ! f1;
> f2;
E.[2,1] - E.[3]
> f1 eq f2;
true
>
> P<x,y,z> := PolynomialRing(R, 3);
> P ! f1;
x^2*y + x^2*z + x*y^2 + 2*x*y*z + x*z^2 + y^2*z + y*z^2
> P ! f2;
x^2*y + x^2*z + x*y^2 + 2*x*y*z + x*z^2 + y^2*z + y*z^2
```

159.2.2 Creation of Symmetric Functions

For each of the 5 different standard basis for symmetric functions, basis elements are indexed via partitions (weakly decreasing positive sequences). The *weight* of a partition is the sum of its entries, and gives the degree of the element.

For example, $[3, 1]$ is a partition of weight 4, hence a symmetric function basis element corresponding to $[3, 1]$ will be a symmetric polynomial of degree 4.

General symmetric functions are linear combinations of basis elements and can be created by taking such linear combinations or via coercion from either another basis or directly from a polynomial.

A . P: AlgSym, [RngIntElt] -> AlgSymElt

Given a partition P, which is a weakly decreasing positive sequence of integers, return the basis element of the algebra of symmetric functions A corresponding to P.

A . i: AlgSym, RngIntElt -> AlgSymElt

Given a positive integer i, return the basis element of the algebra of symmetric functions A corresponding to the partition $[i]$.

Example: Symmetric Functions as Linear Combinations (ex-a9791f)

We can create elements via linear combinations of basis elements.

```
> R := Rationals();
> M := SFAMonomial(R);
> M.[3,1];
M.[3,1]
>  M.4;
```

(continues on next page)

(continued from previous page)

```
M.[4]
> 3 * M.[3,1] - 1/2 * M.4;
3*M.[3,1] - 1/2*M.[4]
```

`A ! f: AlgSym, RngMPolElt -> AlgSymElt`

Given a multivariate polynomial f which is symmetric in all of its indeterminates (and hence is a symmetric function), return f as an element of the algebra of symmetric functions A. This element returned will be expressed in terms of the symmetric function basis of A.

Note that polynomials are considered to be identified with their corresponding representations using the monomial symmetric functions, so the above coercion may involve a change of basis from the monomial symmetric functions to the symmetric function basis used by A. This might appear to introduce unexpected "extra" terms (see the second example below).

Example: Coercing Symmetric Polynomials (ex-6ef1d0)

Symmetric polynomials are symmetric functions. These can be coerced into algebras of symmetric functions and hence expressed in terms of the relevant basis. Polynomials which are not symmetric cannot be coerced.

```
> R := Rationals();
> M := SFAMonomial(R);
> P<[x]> := PolynomialRing(R, 3);
> f := -3*x[1]^3 + x[1]^2*x[2] + x[1]^2*x[3] + x[1]*x[2]^2 + x[1]*x[3]^2 -
>          3*x[2]^3 + x[2]^2*x[3] + x[2]*x[3]^2 - 3*x[3]^3;
> M!f;
M.[2,1] - 3*M.[3]
> M ! (x[1] + x[2]*x[3]);

>> M ! (x[1] + x[2]*x[3]);
     ^
Runtime error in '!': Polynomial is not symmetric
```

Example: Redundant Terms after Coercion (ex-dff88d)

A symmetric polynomial is identified with its representation using a monomial basis. (This ensures consistency when coercing through multiple algebras.) This may cause apparently redundant terms to appear after a coercion to an algebra using a different basis.

```
> R := Integers();
> M := SFAMonomial(R);
> S := SFASchur(R);
> P<X, Y> := PolynomialRing(R, 2);
> f := X^2*Y + X*Y^2;
> M!f;
M.[2,1]
> S!f;
 - 2*S.[1,1,1] + S.[2,1]
```

When we coerce S.[1,1,1] back into our polynomial ring, we see that it evaluates to zero and may seem redundant:

```
> P!S.[1,1,1];
0
```

However, when we use a polynomial ring with more variables we can see the important of that term:

```
> P<X, Y, Z> := PolynomialRing(R, 3);
> P!M.[2,1];
X^2*Y + X^2*Z + X*Y^2 + X*Z^2 + Y^2*Z + Y*Z^2
> P!S.[2,1];
X^2*Y + X^2*Z + X*Y^2 + 2*X*Y*Z + X*Z^2 + Y^2*Z + Y*Z^2
> P!(S.[2,1] - 2*S.[1,1,1]);
X^2*Y + X^2*Z + X*Y^2 + X*Z^2 + Y^2*Z + Y*Z^2
```

`A ! r: AlgSym, RngElt -> AlgSymElt`

Create the scalar element r in the symmetric function algebra A.

Example: Scalar Symmetric Functions (ex-e44a43)

```
> R := Rationals();
> P := SFAPower(R);
> m := P!3;
> m;
3
> Parent(m);
Symmetric Algebra over Rational Field, Power sum symmetric functions as basis
> m + P.[3,2];
3 + P.[3,2]
```

`A ! m: AlgSym, AlgSymElt -> AlgSymElt`

Given a symmetric function algebra A and a symmetric function m (possibly with a different basis), return the element m expressed in terms of the basis of A.

Example: Changing the Basis of a Symmetric Function (ex-ff09c7)

A symmetric function can easily be expressed in terms of any of the symmetric function bases, or as a polynomial.

```
> R := Rationals();
> S := SFASchur(R);
> H := SFAHomogeneous(R);
> P := SFAPower(R);
> E := SFAElementary(R);
> M := SFAMonomial(R);
>
> m := S.[3,1];
> S!m;
S.[3,1]
> H!m;
H.[3,1] - H.[4]
> P!m;
1/8*P.[1,1,1,1] + 1/4*P.[2,1,1] - 1/8*P.[2,2] - 1/4*P.[4]
> E!m;
E.[2,1,1] - E.[3,1] - E.[2,2] + E.[4]
> M!m;
3*M.[1,1,1,1] + 2*M.[2,1,1] + M.[3,1] + M.[2,2]
>
> PP<[x]> := PolynomialRing(R, 4);
> PP ! m;
x[1]^3*x[2] + x[1]^3*x[3] + x[1]^3*x[4] + x[1]^2*x[2]^2 + 2*x[1]^2*x[2]*x[3] +
    2*x[1]^2*x[2]*x[4] + x[1]^2*x[3]^2 + 2*x[1]^2*x[3]*x[4] + x[1]^2*x[4]^2 +
    x[1]*x[2]^3 + 2*x[1]*x[2]^2*x[3] + 2*x[1]*x[2]^2*x[4] + 2*x[1]*x[2]*x[3]^2 +
    3*x[1]*x[2]*x[3]*x[4] + 2*x[1]*x[2]*x[4]^2 + x[1]*x[3]^3 +
    2*x[1]*x[3]^2*x[4] + 2*x[1]*x[3]*x[4]^2 + x[1]*x[4]^3 + x[2]^3*x[3] +
```

(continues on next page)

(continued from previous page)

```
    x[2]^3*x[4] + x[2]^2*x[3]^2 + 2*x[2]^2*x[3]*x[4] + x[2]^2*x[4]^2 +
    x[2]*x[3]^3 + 2*x[2]*x[3]^2*x[4] + 2*x[2]*x[3]*x[4]^2 + x[2]*x[4]^3 +
    x[3]^3*x[4] + x[3]^2*x[4]^2 + x[3]*x[4]^3
```

Example: Homogeneous Functions as Sums of Monomial Functions (ex-16a913)

We show that the k-th homogeneous symmetric function is defined as the sum over all monomial symmetric functions indexed by partitions of weight k.

```
> R := Rationals();
> M := SFA(R : Basis := "Monomial");
> H := SFA(R : Basis := "Homogeneous");
>
> H ! (M.[1]);
H.[1]
> H ! (M.[2] + M.[1,1]);
H.[2]
> H ! (M.[3] + M.[2,1] + M.[1,1,1]);
H.[3]
> H ! (M.[4] + M.[3,1] + M.[2,2] + M.[2,1,1] + M.[1,1,1,1]);
H.[4]
>
> k := 5;
> H ! &+ [ M.P : P in Partitions(k)];
H.[5]
> k := 10;
> H ! &+ [ M.P : P in Partitions(k)];
H.[10]
```

159.3 Structure Operations

159.3.1 Related Structures

The main structure related to an algebra of symmetric functions is its coefficient ring. An algebra of symmetric functions belongs to the Magma category `AlgSym`.

`BaseRing(L): AlgSym -> Rng`

`CoefficientRing(L): AlgSym -> Rng`

Return the coefficient ring of an algebra of symmetric functions L.

`Category(L): AlgSym -> Cat`

`Parent(L): AlgSym -> Pow`

`PrimeRing(L): AlgSym -> Rng`

159.3.2 Ring Predicates and Booleans

The usual ring functions returning boolean values are available on algebras of symmetric functions.

`IsCommutative(L): AlgSym -> BoolElt`

`IsUnitary(L): AlgSym -> BoolElt`

`IsFinite(L): AlgSym -> BoolElt`

`IsOrdered(L): AlgSym -> BoolElt`

`IsField(L): AlgSym -> BoolElt`

`IsEuclideanDomain(L): AlgSym -> BoolElt`

IsPID(L): AlgSym -> BoolElt

IsUFD(L): AlgSym -> BoolElt

IsDivisionRing(L): AlgSym -> BoolElt

IsEuclideanRing(L): AlgSym -> BoolElt

IsDomain(L): AlgSym -> BoolElt

IsPrincipalIdealRing(L): AlgSym -> BoolElt

L eq M: AlgSym, AlgSym -> BoolElt

L ne M: AlgSym, AlgSym -> BoolElt

Return true if L and M are equal (respectively, not equal). Magma considers two algebras to be equal if they are over the same ring.

159.3.3 Predicates on Basis Types

HasSchurBasis(A): AlgSym -> BoolElt

Returns true if A is an algebra with a Schur basis.

HasHomogeneousBasis(A): AlgSym -> BoolElt

HasElementaryBasis(A): AlgSym -> BoolElt

HasPowerSumBasis(A): AlgSym -> BoolElt

HasMonomialBasis(A): AlgSym -> BoolElt

Returns true if A is an algebra with a homogeneous, elementary, power sum, or monomial basis respectively.

159.4 Element Operations

159.4.1 Parent and Category

The category for elements in algebras of symmetric functions is AlgSymElt.

Parent(f): AlgSymElt -> AlgSym

Category(f): AlgSymElt -> Cat

159.4.2 Print Styles

By default, elements are printed using a lexicographical ordering on the partitions indexing the basis elements. For partitions of the same weight w this ordering is the reverse of that in Partitions(w). More generally, basis elements indexed by partitions with smaller entries print first (see Testing Order Relations). It is possible to rearrange the linear combination of basis elements so that the basis elements indexed by the longest partition print first and also so that the basis elements indexed by the partition with greatest maximal part (first entry) print first.

A ` PrintStyle: AlgSym -> MonStgElt

Retrieve or set the style in which elements of the algebra A will print. The default is the lexicographical ordering described above, "Lex". Other options are "Length" and "MaximalPart".

Example: Setting the Print Style (ex-17b218)

We demonstrate the setting of the PrintStyle and the corresponding printing of elements.

```
> M := SFAMonomial(Rationals());
> M`PrintStyle;
Lex
> P := Partitions(3);
> P;
[
```

(continues on next page)

(continued from previous page)

```
    [ 3 ],
    [ 2, 1 ],
    [ 1, 1, 1 ]
]
> f := &+[M.p : p in P];
> f;
M.[1,1,1] + M.[2,1] + M.[3]
> M`PrintStyle := "Length";
> f;
M.[3] + M.[2,1] + M.[1,1,1]
> M`PrintStyle := "MaximalPart";
> f;
M.[1,1,1] + M.[2,1] + M.[3]
```

159.4.3 Additive Arithmetic Operators

The usual unary and binary ring element operations are available for symmetric functions. It is possible to combine elements of different algebras (so long as the coefficient rings are compatible) in these operations. Where the elements are written with respect to different bases, the result will be written with respect to the basis of the second operand.

`+ a: AlgSymElt -> AlgSymElt`

`- a: AlgSymElt -> AlgSymElt`

`a + b: AlgSymElt, AlgSymElt -> AlgSymElt`

`a - b: AlgSymElt, AlgSymElt -> AlgSymElt`

`a +:= b: AlgSymElt, AlgSymElt -> AlgSymElt`

`a -:= b: AlgSymElt, AlgSymElt -> AlgSymElt`

159.4.4 Multiplication

Where the elements being multiplied are expressed with respect to different bases, the result will again be expressed with respect to the basis of the second operand. The algorithm used for multiplication is dependent on the bases with respect to which the elements are expressed.

`a * b: AlgSymElt, AlgSymElt -> AlgSymElt`

`a *:= b: AlgSymElt, AlgSymElt -> AlgSymElt`

The product of the symmetric functions a and b.

If b is expressed with respect to a Schur basis and a is expressed with respect to a power sum, elementary, or monomial basis then the algorithm based on Muirs rule [Mui60] is used. If a is expressed with respect to a homogeneous basis the algorithm based on the Pieri rule [Mac95] is used and if a is also expressed with respect to a Schur basis then the method of Schubert polynomials [LS85] is used.

Special algorithms are also used if b is expressed with respect to a monomial basis and a is expressed with respect to a homogeneous, elementary, or monomial basis.

When both a and b have a homogeneous, elementary, or power sum basis, the multiplication of basis elements involves the merging of the parts of the partitions. This follows from the definition of the basis elements $f_\lambda = \prod_i f_{\lambda_i}$ where f_λ is a homogeneous, elementary, or power sum basis element.

Otherwise a is coerced into the parent of b before the elements are multiplied.

The degree of the result is the sum of the degrees of both the operands.

`a ^ k: AlgSymElt, RngIntElt -> AlgSymElt`

Example: Multiplication of Symmetric Functions (ex-cf1c1c)

```
> Q := Rationals();
> s := SFASchur(Q);
> m := SFAMonomial(Q);
> m.[3]*s.[2,1];
s.[2,1,1,1,1] + s.[5,1] - s.[2,2,2] - s.[3,3]
```

To illustrate the merging of the partitions :

```
> E := SFAElementary(Q);
> E.4*E.3*E.1;
E.[4,3,1]
```

This is the definition of $e_{[4,3,1]}$.

159.4.5 Plethysm

Plethysm is also referred to as composition of symmetric functions.

`a ~ b: AlgSymElt, AlgSymElt -> AlgSymElt`

This operator computes the plethysm or composition of the symmetric functions a and b. The result is given with respect to the basis of the second operand.

The degree of the result is the product of the degrees of operands which may be very large.

Example: Plethysm of Symmetric Functions (ex-107368)

```
> Q := Rationals();
> s := SFASchur(Q);
> m := SFAMonomial(Q);
> m.[3]~s.[2,1];
s.[4,1,1,1,1,1] - s.[3,2,1,1,1,1] - s.[5,1,1,1,1] + s.[2,2,2,1,1,1] +
s.[3,3,1,1,1] + s.[6,1,1,1] - s.[2,2,2,2,1] - s.[3,3,2,1] - s.[6,2,1] +
s.[4,4,1] + s.[3,2,2,2] + s.[5,2,2] - s.[4,3,2] + 2*s.[3,3,3] + s.[6,3] -
s.[5,4]
```

159.4.6 Boolean Operators

`IsHomogeneous(s): AlgSymElt -> BoolElt`

Returns `true` if the partitions indexing the basis elements present in the symmetric function s are all of the same weight. This implies that each term has the same degree so is the same as the polynomial expansion of s being a homogeneous polynomial.

`IsZero(s): AlgSymElt -> BoolElt`

`IsOne(s): AlgSymElt -> BoolElt`

`IsMinusOne(s): AlgSymElt -> BoolElt`

`s eq t: AlgSymElt, AlgSymElt -> BoolElt`

`s ne t: AlgSymElt, AlgSymElt -> BoolElt`

Return `true` if the symmetric functions s and t are (not) the same.

159.4.7 Accessing Elements

`Coefficient(s, p): AlgSymElt, SeqEnum -> RngElt`

Given a symmetric function s return the coefficient of the basis element A_p, where A is the parent of s and p is a sequence defining a partition. The coefficient may be zero.

Support(s): AlgSymElt -> [SeqEnum], [RngElt]

Return two parallel sequences of the partitions indexing the basis elements and the coefficients of those basis elements in the symmetric function s, which is a linear combination of basis elements.

Length(s): AlgSymElt -> RngIntElt

Given a symmetric function s, return the length of s, i.e., the number of basis elements having non zero coefficients in s, with respect to the current basis.

Example: Decomposing and Rebuilding an Element (ex-4a300f)

We pull apart an element and show that we can put it back together again.

```
> H := SFAHomogeneous(Rationals());
> P := Partitions(4);
> f := &+[Random(1, 5)*H.p : p in P];
> f;
H.[1,1,1,1] + 4*H.[2,1,1] + 4*H.[2,2] + 5*H.[3,1] + 5*H.[4]
> s, e := Support(f);
> s, e;
[
    [ 1, 1, 1, 1 ],
    [ 2, 1, 1 ],
    [ 2, 2 ],
    [ 3, 1 ],
    [ 4 ]
]
[ 1, 4, 4, 5, 5 ]
> f eq &+[e[i]*H.s[i] : i in [1 .. Length(f)]];
true
```

Degree(s): AlgSymElt -> RngIntElt

Given a symmetric function s, return the degree of s, i.e., the maximal degree of the basis elements having non zero coefficients in s, which is the maximal weight of the partitions indexing those basis elements.

159.4.8 Multivariate Polynomials

A symmetric function may be seen as a polynomial in any number of variables.

P ! s: RngMPol, AlgSymElt -> RngMPolElt

Return the polynomial expansion of the symmetric function s in the polynomial ring P.

Example: Restricting to Finitely Many Variables (ex-10b1c4)

```
> S := SFASchur(GF(7));
> s := S.[3,1];
```

Now we compute the multivariate polynomial we get by restricting to 5 variables.

```
> G<e1, e2, e3, e4, e5> := PolynomialRing(GF(7), 5);
> p := G!s;
> p;
e1^3*e2 + e1^3*e3 + e1^3*e4 + e1^3*e5 + e1^2*e2^2 + 2*e1^2*e2*e3 + 2*e1^2*e2*e4
    + 2*e1^2*e2*e5 + e1^2*e3^2 + 2*e1^2*e3*e4 + 2*e1^2*e3*e5 + e1^2*e4^2 +
    2*e1^2*e4*e5 + e1^2*e5^2 + e1*e2^3 + 2*e1*e2^2*e3 + 2*e1*e2^2*e4 +
    2*e1*e2^2*e5 + 2*e1*e2*e3^2 + 3*e1*e2*e3*e4 + 3*e1*e2*e3*e5 + 2*e1*e2*e4^2 +
    3*e1*e2*e4*e5 + 2*e1*e2*e5^2 + e1*e3^3 + 2*e1*e3^2*e4 + 2*e1*e3^2*e5 +
    2*e1*e3*e4^2 + 3*e1*e3*e4*e5 + 2*e1*e3*e5^2 + e1*e4^3 + 2*e1*e4^2*e5 +
    2*e1*e4*e5^2 + e1*e5^3 + e2^3*e3 + e2^3*e4 + e2^3*e5 + e2^2*e3^2 +
```

(continues on next page)

(continued from previous page)

```
    2*e2^2*e3*e4 + 2*e2^2*e3*e5 + e2^2*e4^2 + 2*e2^2*e4*e5 + e2^2*e5^2 + e2*e3^3
    + 2*e2*e3^2*e4 + 2*e2*e3^2*e5 + 2*e2*e3*e4^2 + 3*e2*e3*e4*e5 + 2*e2*e3*e5^2
    + e2*e4^3 + 2*e2*e4^2*e5 + 2*e2*e4*e5^2 + e2*e5^3 + e3^3*e4 + e3^3*e5 +
    e3^2*e4^2 + 2*e3^2*e4*e5 + e3^2*e5^2 + e3*e4^3 + 2*e3*e4^2*e5 + 2*e3*e4*e5^2
    + e3*e5^3 + e4^3*e5 + e4^2*e5^2 + e4*e5^3
```

To check the polynomial is actually symmetric, we can use the Magma intrinsic `IsSymmetric`, which also computes an expansion as a sum of elementary symmetric polynomials.

```
> IsSymmetric(p,G);
true e1^2*e2 + 6*e1*e3 + 6*e2^2 + e4
```

Which is identical to the result of

```
> E := SFAElementary(GF(7));
> E!s;
E.[2,1,1] + 6*E.[2,2] + 6*E.[3,1] + E.[4]
```

Example: Change of Alphabet (ex-86835b)

These conversions may be used to change the alphabet of a symmetric function. For example, if we substitute the variable x_i by $x_i + 1$, the result is again a symmetric function, however we use polynomials to do the evaluation.

```
> S := SFASchur(Rationals());
> R<a, b, c, d, e> := PolynomialRing(Rationals(), 5);
> p := Polynomial(S.[3,2], R);
> q := Evaluate(p, [a+1, b+1, c+1, d+1, e+1]);
> x, y := IsCoercible(S, q);
> y;
175 + 175*S.[1] + 70*S.[1,1] + 70*S.[2] + 35*S.[2,1] + 7*S.[2,2] + 10*S.[3] +
5*S.[3,1] + S.[3,2]
```

159.4.9 Frobenius Homomorphism

There is an automorphism which maps the elementary symmetric function to the homogeneous symmetric function.

`Frobenius(s): AlgSymElt -> AlgSymElt`

Given any symmetric function s, compute the image of s under the Frobenius automorphism. The image will have the same parent as s. When the power sum symmetric functions are involved, it may be necessary to work with a coefficient ring which allows division by an integer.

Example: Frobenius Automorphism (ex-41dc58)

It is known that the Frobenius automorphism on the Schur functions acts just by conjugating the indexing partitions.

```
> S := SFASchur(Integers());
> E := SFAElementary(Integers());
> h := S!E.[3,3,3];
> h;
S.[1,1,1,1,1,1,1,1,1] + 2*S.[2,1,1,1,1,1,1,1] + S.[3,1,1,1,1,1] +
3*S.[2,2,1,1,1,1,1] + 2*S.[3,2,1,1,1,1] + 4*S.[2,2,2,1,1,1] + S.[3,3,1,1,1] +
3*S.[3,2,2,1,1] + 2*S.[2,2,2,2,1] + 2*S.[3,3,2,1] + S.[3,2,2,2] + S.[3,3,3]
```

Now apply the Frobenius automorphism.

```
> f:=Frobenius(h);
> f;
S.[3,3,3] + 2*S.[4,3,2] + S.[4,4,1] + S.[5,2,2] + 3*S.[5,3,1] + 2*S.[5,4] +
2*S.[6,2,1] + 4*S.[6,3] + S.[7,1,1] + 3*S.[7,2] + 2*S.[8,1] + S.[9]
```

To check whether the coefficient of the basis element indexed by a partition in one element is the same as the coefficient of the basis element indexed by the conjugate partition in the other :

```
> p:=Partitions(f);
> for pp in p do
>    if Coefficient(h, ConjugatePartition(pp)) ne Coefficient(f, pp) then
>       print pp;
>    end if;
> end for;
```

159.4.10 Inner Product

An inner product on Λ is defined by

$$\langle m_\lambda, h_{\lambda'} \rangle = \delta_{\lambda,\lambda'}.$$

This definition ensures that the bases m_λ and h_λ are dual to each other. This is the inner product used by Magma.

`InnerProduct(a, b): AlgSymElt, AlgSymElt -> RngElt`

Computes the inner product of the symmetric functions a and b.

Example: Inner Product of Symmetric Functions `(ex-71d4ca)`

The inner product of a elementary symmetric function indexed by a partition and the homogeneous symmetric function indexed by the conjugate partition is 1. (This allows the computation of irreducible representations of the symmetric group.) To check this with Magma, we do the following:

```
> E := SFAElementary(Rationals());
> H := SFAHomogeneous(Rationals());
> p:=RandomPartition(45);
> pc:=ConjugatePartition(p);
> InnerProduct(E.p,H.pc);
1
```

As it should be the result is 1.

159.4.11 Combinatorial Objects

The Schur function is the generating function of standard tableaux. Therefore, it is possible to get the corresponding tableaux.

`Tableaux(sf, m): AlgSymElt, RngIntElt -> SetMulti`

Given a Schur function sf over the integers with positive coefficients, return the multiset of the tableaux, with maximal entry m, for which sf is the generating function.

159.4.12 Symmetric Group Character

A Schur function indexed by a single partition, i.e. a basis element, corresponds to an irreducible character of the symmetric group.

`SymmetricCharacter(sf): AlgSymElt -> AlgChtrElt`

Given an element sf of an algebra of symmetric functions return a linear combination of irreducible characters of the symmetric group, whose coefficients are the coefficients of sf with respect to the Schur function basis.

Example: Characters Induced from Young Subgroups (ex-102778)

We look at a result in the representation theory of the symmetric group. There is exactly one irreducible character contained in both the induced character from the identity character of a Young subgroup indexed by the partition I and the induced character from the alternating character of a Young subgroup indexed by the partition J, conjugate to I. As the induced character of the identity character of a Young subgroup corresponds to the homogeneous symmetric function h_I and the induced character of the alternating character corresponds to the elementary symmetric function e_J, we can verify this using the following routine:

```
> H := SFAHomogeneous(Rationals());
> E := SFAElementary(Rationals());
> p := Partitions(7);
> for I in p do
>     a := SymmetricCharacter(H.I);
>     J := ConjugatePartition(I);
>     b := SymmetricCharacter(E.J);
>     i := InnerProduct(a,b);
>     if i ne 1 then print i; end if;
> end for;
```

And there should be no output.

159.4.13 Restrictions

It is possible to form symmetric functions whose support is a subset of the support of a given symmetric function, subject to some restrictions.

`RestrictDegree(a, n): AlgSymElt, RngIntElt -> AlgSymElt`

```
Exact: BoolElt                    Default: true
```

Return the symmetric function which is a linear combination of those basis elements of the symmetric function a with degree n. This is the restriction of a into the submodule Λ^n. If `Exact` is `false` then the basis elements included will have degree $\leq n$. This is the restriction of a into $\bigcup_{k \leq n} \Lambda^k$.

`RestrictPartitionLength(a, n): AlgSymElt, RngIntElt -> AlgSymElt`

```
Exact: BoolElt                    Default: true
```

Return the symmetric function which is a linear combination of those basis elements of the symmetric function a whose indexing partitions are of length n. If `Exact` is `false`, then the indexing partitions will be of length $\leq n$.

`RestrictParts(a, n): AlgSymElt, RngIntElt -> AlgSymElt`

```
Exact: BoolElt                    Default: true
```

Return the symmetric function which is a linear combination of those basis elements of the symmetric function a whose indexing partitions have maximal part n. If `Exact` is `false`, then the indexing partitions will have maximal part $\leq n$.

159.5 Transition Matrices

One area of interest in the theory of symmetric functions is the study of the change of bases between the five different bases. The matrices of change of base between the s_λ, h_λ, m_λ, and the e_λ are all integer matrices. Only when changing from one of these four bases to the p_λ, is the matrix over the rationals. For a discussion of their computation and interactions see [Mac95, pages 54--58].

In Magma there are routines available to obtain all these matrices. These routines are described below. Some interactions of these matrices are verified in examples.

159.5.1 Transition Matrices from Schur Basis

SchurToMonomialMatrix(n): RngIntElt -> AlgMatElt

Computes the matrix for the expansion of a Schur function indexed by a partition of weight n as a sum of monomial symmetric functions. This matrix is also known as the table of *Kostka* numbers. These are the numbers of tableaux of each shape and content. The content of a tableau prescribes its entries, so for example a tableau with content $[2, 1, 4, 1]$ has two 1's, one 2, four 3's, and one 4. The entries of the matrix are non negative integers. The matrix is upper triangular.

Example: Kostka Numbers (ex-c92c1f)

Compute the base change matrix from the Schur functions to the monomial symmetric functions for degree 5. The entries in this matrix are what are known as the *Kostka* numbers. They count the number of young tableaux on a given shape Look on the order of the labelling partitions, and check whether the entry 3 in the upper right corner is right by generating the corresponding tableaux.

```
> M := SchurToMonomialMatrix(5);
> M;
[1 1 1 1 1 1 1]
[0 1 1 2 2 3 4]
[0 0 1 1 2 3 5]
[0 0 0 1 1 3 6]
[0 0 0 0 1 2 5]
[0 0 0 0 0 1 4]
[0 0 0 0 0 0 1]
> Parts := Partitions(5);
> Parts;
[
    [ 5 ],
    [ 4, 1 ],
    [ 3, 2 ],
    [ 3, 1, 1 ],
    [ 2, 2, 1 ],
    [ 2, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1 ]
]
> #TableauxOnShapeWithContent(Parts[2], Parts[6]);
3
> M[2,6];
3
```

SchurToHomogeneousMatrix(n): RngIntElt -> AlgMatElt

Computes the matrix for the expansion of a Schur function indexed by a partition of weight n as a sum of homogeneous symmetric functions. The entries of the matrix are positive and negative integers. The matrix is lower triangular. It is the transpose of the matrix returned by MonomialToSchurMatrix(n).

SchurToPowerSumMatrix(n): RngIntElt -> AlgMatElt

Computes the matrix for the expansion of a Schur function indexed by a partition of weight n as a sum of power sum symmetric functions. The entries of the matrix are rationals.

SchurToElementaryMatrix(n): RngIntElt -> AlgMatElt

Computes the matrix for the expansion of a Schur function indexed by a partition of weight n as a sum of elementary symmetric functions. The entries of the matrix are positive and negative integers. The matrix is upper left triangular.

Example: Transition Matrices Agree with Coercion (ex-ed1ff8)

We verify by hand that the action of base change matrix is the same as coercion.

```
> S := SFASchur(Rationals());
> P := SFAPower(Rationals());
> NumberOfPartitions(4);
5
> m := SchurToPowerSumMatrix(4);
> Partitions(4);
[
    [ 4 ],
    [ 3, 1 ],
    [ 2, 2 ],
    [ 2, 1, 1 ],
    [ 1, 1, 1, 1 ]
]
> s := S.[3, 1] + 5*S.[1, 1, 1, 1] - S.[4];
> s;
5*S.[1,1,1,1] + S.[3,1] - S.[4]
> p, c := Support(s);
> c;
[ 5, 1, -1 ]
> p;
[
    [ 1, 1, 1, 1 ],
    [ 3, 1 ],
    [ 4 ]
]
> cm := Matrix(Rationals(), 1, 5, [-1, 1, 0, 0, 5]);
> cm*m;
[-7/4  4/3  3/8 -5/4 7/24]
> P!s;
7/24*P.[1,1,1,1] - 5/4*P.[2,1,1] + 3/8*P.[2,2] + 4/3*P.[3,1] - 7/4*P.[4]
```

The coefficients of the coerced element are the reverse of the matrix product, consistent with the partition in the coerced element being in reverse order to those in `Partitions(4)`.

159.5.2 Transition Matrices from Monomial Basis

`MonomialToSchurMatrix(n): RngIntElt -> AlgMatElt`

Computes the matrix for the expansion of a monomial symmetric function indexed by a partition of weight n as a sum of Schur symmetric functions. The entries of the matrix are positive and negative integers. The matrix is upper triangular. It is the transpose of the matrix returned by `SchurToHomogeneousMatrix(n)`.

`MonomialToHomogeneousMatrix(n): RngIntElt -> AlgMatElt`

Computes the matrix for the expansion of a monomial symmetric function indexed by a partition of weight n as a sum of homogeneous symmetric functions. The entries are positive and negative integers.

`MonomialToPowerSumMatrix(n): RngIntElt -> AlgMatElt`

Computes the matrix for the expansion of a monomial symmetric function indexed by a partition of weight n as a sum of power sum symmetric functions. The entries are rationals. The matrix is lower triangular.

`MonomialToElementaryMatrix(n): RngIntElt -> AlgMatElt`

Computes the matrix for the expansion of a monomial symmetric function indexed by a partition of weight n as a sum of elementary symmetric functions. The entries of the matrix are positive and negative integers. The matrix is upper left triangular.

159.5.3 Transition Matrices from Homogeneous Basis

`HomogeneousToSchurMatrix(n): RngIntElt -> AlgMatElt`

Computes the matrix for the expansion of a homogeneous symmetric function indexed by a partition of weight n as a sum of Schur symmetric functions. The entries of the matrix are positive integers. The matrix is lower triangular.

Example: Homogeneous-to-Schur Matrix as a Transpose (ex-eb6dd0)

It is known that the matrix computed by `HomogeneousToSchurMatrix` is the transpose of the matrix computed by `SchurToMonomialMatrix`.

```
> SchurToMonomialMatrix(7) eq Transpose(HomogeneousToSchurMatrix(7));
true
```

`HomogeneousToMonomialMatrix(n): RngIntElt -> AlgMatElt`

Computes the matrix M for the expansion of a homogeneous symmetric function indexed by a partition of weight n as a sum of monomial symmetric functions. The entries of the matrix are positive integers. The matrix has no zero entries.

The coefficient $M_{\mu,\lambda}$ in the expansion $h_\lambda = \sum_\mu M_{\mu,\lambda} m_\mu$ is the number of non negative integer matrices with row sums λ_i and column sums μ_j, see [Mac95, page 57].

Example: Symmetry of the Homogeneous-to-Monomial Matrix (ex-5582be)

The matrix converting from homogeneous basis to monomial basis is symmetric.

```
> IsSymmetric(HomogeneousToMonomialMatrix(7));
true
```

`HomogeneousToPowerSumMatrix(n): RngIntElt -> AlgMatElt`

Computes the matrix for the expansion of a homogeneous symmetric function indexed by a partition of weight n as a sum of power sum symmetric functions. The entries of the matrix are positive rationals. The matrix is upper triangular.

`HomogeneousToElementaryMatrix(n): RngIntElt -> AlgMatElt`

Computes the matrix for the expansion of a homogeneous symmetric function indexed by a partition of weight n as a sum of elementary symmetric functions. The entries of the matrix are integers. The matrix is upper triangular.

Example: Homogeneous-to-Elementary Transition Matrix (ex-6bb56e)

It is known that the matrix compute by `HomogeneousToElementaryMatrix` is the same as the matrix computed by `ElementaryToHomogeneousMatrix`.

```
> HomogeneousToElementaryMatrix(7) eq ElementaryToHomogeneousMatrix(7);
true
```

159.5.4 Transition Matrices from Power Sum Basis

`PowerSumToSchurMatrix(n): RngIntElt -> AlgMatElt`

Computes the matrix for the expansion of a power sum symmetric function indexed by a partition of weight n as a sum of Schur symmetric functions. The entries of the matrix are positive and negative integers. This matrix is the character table of the symmetric group.

Example: Character Table of the Symmetric Group (ex-75ee8a)

The matrix returned by `PowerSumToSchurMatrix` is compared to the character table of the appropriate symmetric group.

```
> PowerSumToSchurMatrix(5);
[ 1 -1  0  1  0 -1  1]
```

(continues on next page)

(continued from previous page)

```
[ 1  0 -1  0  1  0 -1]
[ 1 -1  1  0 -1  1 -1]
[ 1  1 -1  0 -1  1  1]
[ 1  0  1 -2  1  0  1]
[ 1  2  1  0 -1 -2 -1]
[ 1  4  5  6  5  4  1]
> CharacterTable(Sym(5));


Character Table
---------------



------------------------------
Class |   1  2  3  4  5  6  7
Size  |   1 10 15 20 30 24 20
Order |   1  2  2  3  4  5  6
------------------------------
p  =  2   1  1  1  4  3  6  4
p  =  3   1  2  3  1  5  6  2
p  =  5   1  2  3  4  5  1  7
------------------------------
X.1   +   1  1  1  1  1  1  1
X.2   +   1 -1  1  1 -1  1 -1
X.3   +   4  2  0  1  0 -1 -1
X.4   +   4 -2  0  1  0 -1  1
X.5   +   5  1  1 -1 -1  0  1
X.6   +   5 -1  1 -1  1  0 -1
X.7   +   6  0 -2  0  0  1  0
```

In the character table the first row is the unity character, which corresponds to the first column of the transition matrix. The second row of the character table is the alternating character which corresponds to the last column of the transition matrix. The first column of the character table contains the dimensions of the irreducible characters, this is the last row of the transition matrix.

PowerSumToMonomialMatrix(n): RngIntElt -> AlgMatElt

Computes the matrix for the expansion of a power sum symmetric function indexed by a partition of weight n as a sum of monomial symmetric functions. The entries of the matrix are positive integers. The matrix is lower triangular.

PowerSumToHomogeneousMatrix(n): RngIntElt -> AlgMatElt

Computes the matrix for the expansion of a power sum symmetric function indexed by a partition of weight n as a sum of homogeneous symmetric functions. The entries of the matrix are integers. The matrix is upper triangular.

PowerSumToElementaryMatrix(n): RngIntElt -> AlgMatElt

Computes the matrix for the expansion of a power sum symmetric function indexed by a partition of weight n as a sum of elementary symmetric functions. The entries of the matrix are integers. The matrix is upper triangular.

159.5.5 Transition Matrices from Elementary Basis

ElementaryToSchurMatrix(n): RngIntElt -> AlgMatElt

Computes the matrix for the expansion of an elementary symmetric function indexed by a partition of weight n as a sum of Schur symmetric functions. The entries of the matrix are positive integers.

ElementaryToMonomialMatrix(n): RngIntElt -> AlgMatElt

Computes the matrix M for the expansion of an elementary symmetric function indexed by a

partition of weight n as a sum of monomial symmetric functions. The entries of the matrix are positive integers.

The coefficient $M_{\mu,\lambda}$ in the expansion $e_\lambda = \sum_\mu M_{\mu,\lambda} m_\mu$ is the number of 0-1 integer matrices with row sum λ_i and column sum μ_j, see [Mac95, page 57].

Example: Symmetry of the Elementary-to-Monomial Matrix `(ex-5a0537)`

The matrix converting from elementary basis to monomial basis is symmetric.

```
> IsSymmetric(ElementaryToMonomialMatrix(7));
true
```

`ElementaryToHomogeneousMatrix(n): RngIntElt -> AlgMatElt`

Computes the matrix for the expansion of a elementary symmetric function indexed by a partition of weight n as a sum of homogeneous symmetric functions.

`ElementaryToPowerSumMatrix(n): RngIntElt -> AlgMatElt`

Computes the matrix for the expansion of a elementary symmetric function indexed by a partition of weight n as a sum of power sum symmetric functions.

159.6 Bibliography

- [LS85] Alain Lascoux and Marcel-Paul Schützenberger. Schubert polynomials and the Littlewood-Richardson rule. *Lett. Math. Phys.*, 10(2-3):111–124, 1985.
- [Mac95] I. G. Macdonald. *Symmetric functions and Hall polynomials*. The Clarendon Press Oxford University Press, New York, second edition, 1995. ISBN 0-19-853489-2. With contributions by A. Zelevinsky, Oxford Science Publications.
- [Mui60] Thomas Muir. *A treatise on the theory of determinants*. Dover Publications Inc., New York, 1960.

Chapter 160

Incidence Structures and Designs

160.1 Introduction

Since there is some variation between authors of the terminology employed in design theory, we begin with some definitions. An *incidence structure* is a triple $D = (P, B, I)$, where:

(a) P is a set, the elements of which are called *points*;

(b) B is a set, the elements of which are called *blocks*;

(c) I is an incidence relation between P and B, so that $I \subset P \times B$. The elements of I are called *flags*.

Usually, blocks will be subsets of P, so that instead of writing $(p, b) \in I$, we write $p \in b$. In general, repeated blocks are allowed so that different blocks may correspond to the same subset of P. If D has no repeated blocks, then we say that D is *simple*.

An incidence structure D is said to be *uniform* with *blocksize* k if D has at least one block and all blocks contain exactly k points. A uniform incidence structure is called *trivial* if each k-subset of the point set appears as a block (at least once).

Let $t \geq 0$ be an integer. Then an incidence structure D is said to be *t–balanced* if there exists an integer $\lambda \geq 1$ such that each t–subset of the point set is contained in exactly λ blocks of D.

A *near–linear space* is an incidence structure in which every block contains at least two points and any two points lie in at most one block. A *linear space* is a near–linear space in which any two points lie in *exactly* one block. It is usual, when discussing near–linear spaces, to use the term *line* in place of the term *block*.

Let v, k, t, and λ be integers with $v \geq k \geq t \geq 0$ and $\lambda \geq 1$. A *t–design* with v points and *blocksize* k is an incidence structure $D = (P, B, I)$ where:

(a) The cardinality of P is v;

(b) D is uniform with blocksize k;

(c) D is simple;

(d) For each t–subset T of P there are exactly λ blocks of B incident with all the points of T (so D is t–balanced).

Such a design is usually referred to as a t–(v, k, λ) design. The parameter λ is called the *index* of the design. If b denotes the cardinality of B, a t–design with $v = b$ and $t \geq 2$ is called a *symmetric design*. A t–design with $\lambda = 1$ is called a *Steiner design*. A design which is trivial is also called a *complete design*. Note that a design D must contain at least one block (i.e $b > 0$).

The category names for the different families of incidence structures are as follows:

- Incidence structure : `Inc`
- Near-linear space : `IncNsp`

- Linear space : `IncLsp`
- t–design : `Dsgn`

160.2 Construction of Incidence Structures and Designs

`IncidenceStructure< v | X >: RngIntElt, List -> Inc`

`IncidenceStructure< P | X >: SetIndx, List -> Inc`

Construct the incidence structure D having point set $P = \{@p_1, p_2, \ldots, p_v@\}$ (where $p_i = i$ for each i if the first form of the constructor is used), and block set $B = \{B_1, B_2, \ldots, B_b\}$ given by X. The value of X must be either:

(a) A list of subsets of the set P.

(b) A sequence, set, or indexed set of subsets of P.

(c) A list of blocks of an existing incidence structure.

(d) A sequence, set, or indexed set of blocks of an existing incidence structure.

(e) A combination of the above.

(f) A $v \times b$ $(0,1)$-matrix A, where A may be defined over any coefficient ring. The matrix A will be interpreted as the incidence matrix for the incidence structure D.

(g) A set of codewords of a linear code with length v. The block set of D is taken to be the set of supports of the codewords.

The incidence structure D produced by this constructor will have P as its point set and B as its set of blocks. The function returns three values:

(i) The incidence structure D;

(ii) The point-set P for D; and

(iii) The block-set B for D.

`NearLinearSpace< v | X : parameters >: RngIntElt, List -> IncNsp`

`NearLinearSpace< P | X : parameters >: SetIndx, List -> IncNsp`

`Check: BoolElt`	`Default: true`

Construct the near–linear space S on the set of points $P = \{@p_1, p_2, \ldots, p_v@\}$ (where $p_i = i$ for each i if the first form of the constructor is used), with lines $L = \{L_1, L_2, \ldots, L_b\}$ given by X. The value of X must be either:

(a) A list of subsets of the set P.

(b) A sequence, set, or indexed set of subsets of P.

(c) A list of blocks of an existing incidence structure.

(d) A sequence, set, or indexed set of blocks of an existing incidence structure.

(e) A combination of the above.

(f) A $v \times b$ $(0,1)$-matrix A, where A may be over any coefficient ring. The matrix A will be interpreted as the incidence matrix for the near–linear space S.

(g) A set of codewords of a linear code with length v. The line set of S is taken to be the set of supports of the codewords.

The set of lines L defined by X must satisfy two properties:

(a) Each line of L must contain at least two points;

(b) Any two points of P may lie on at most one line.

The optional boolean argument `Check` indicates whether or not to check that these two properties are satisfied.

The near–linear space S produced by this constructor will have P as its point set and L as its set of lines. The function returns three values:

(i) The near–linear space S;

(ii) The point-set P for S; and

(iii) The line-set L for S.

```
LinearSpace< v | X : parameters >: RngIntElt, List -> IncLsp
LinearSpace< P | X : parameters >: SetIndx, List -> IncLsp
```

```
Check: BoolElt                    Default: true
```

Construct the linear space S on the set of points $P = \{@p_1, p_2, \ldots, p_v@\}$ (where $p_i = i$ for each i if the first form of the constructor is used), with lines $L = \{L_1, L_2, \ldots, L_b\}$ given by X. The value of X must be either:

(a) A list of subsets of the set P.

(b) A sequence, set, or indexed set of subsets of P.

(c) A list of blocks of an existing incidence structure.

(d) A sequence, set, or indexed set of blocks of an existing incidence structure.

(e) A combination of the above.

(f) A $v \times b$ $(0,1)$-matrix A, where A may be defined over any coefficient ring. The matrix A will be interpreted as the incidence matrix for the linear space S.

(g) A set of codewords of a linear code with length v. The line set of S is taken to be the set of supports of the codewords.

The set of lines L defined by X must satisfy two properties:

(a) Each line of L must contain at least two points;

(b) Any two points of P must lie on precisely one line.

The optional boolean argument `Check` indicates whether or not to check that these two properties are satisfied.

The linear space S produced by this constructor will have P as its point set and L as its set of lines. The function returns three values:

(i) The linear space S;

(ii) The point-set P for S; and

(iii) The line-set L for S.

```
Design< t, v | X : parameters >: RngIntElt, RngIntElt, List -> Dsgn
Design< t, P | X : parameters >: RngIntElt, SetIndx, List -> Dsgn
```

```
Check: BoolElt                    Default: true
Al   : MonStg                     Default: "NoOrbits"
```

Construct the t–design D on the set of points $P = \{@p_1, p_2, \ldots, p_v@\}$ (where $p_i = i$ for each i if the first form of the constructor is used), with blocks $B = \{B_1, B_2, \ldots, B_b\}$ given by X. The value of X must be either:

(a) A list of subsets of the set P.

(b) A sequence, set, or indexed set of subsets of P.

(c) A list of blocks of an existing incidence structure.

(d) A sequence, set, or indexed set of blocks of an existing incidence structure.

(e) A combination of the above.

(f) A $v \times b$ $(0,1)$-matrix A, where A may be over any coefficient ring. The matrix A will be interpreted as the incidence matrix for the t–(v, k, λ) design D.

(g) A set of codewords of a linear code with length v. The block set of D is taken to be the set of supports of the codewords.

The set of blocks B defined by X must satisfy three properties:

(a) Each block must contain the same number, k say, of points.

(b) Every t–subset of P must lie in the same number, λ say (where $\lambda > 0$), of blocks.

(c) There must be no repeated blocks.

The optional boolean argument `Check` indicates whether or not to check that these three properties are satisfied. The optional parameter `Al` can be used to specify the algorithm used for balance testing, see the introduction to Section *Elementary Properties of Incidence Structures and Designs* for a full description of its usage.

The t–(v, k, λ) design D produced by this constructor will have P as its point set and B as its set of blocks. The function returns three values:

(i) The design D;

(ii) The point-set P for D; and

(iii) The block-set B for D.

Example: Constructing the Fano Plane `(ex-3bf146)`

The Fano plane, considered as a 2-$(7, 3, 1)$ design, may be constructed by the following statement:

```
> F := Design< 2, 7 | {1,2,3}, {1,4,5}, {1,6,7}, {2,4,7},
>                     {2,5,6}, {3,5,7}, {3,4,6} >;
> F: Maximal;
2-(7, 3, 1) Design with 7 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7 @}
Blocks:
    {1, 2, 3},
    {1, 4, 5},
    {1, 6, 7},
    {2, 4, 7},
    {2, 5, 6},
    {3, 5, 7},
    {3, 4, 6}
```

General incidence structures are allowed repeated blocks, as in the following:

```
> S := IncidenceStructure< {@ 4, 5, 7, 9 @} | [{4, 5, 7}, {7, 9},
>                                              {5, 7, 9} , {7, 9}] >;
> S: Maximal;
Incidence Structure on 4 points with 4 blocks
Points: {@ 4, 5, 7, 9 @}
Blocks:
    {4, 5, 7},
    {7, 9},
    {5, 7, 9},
    {7, 9}
```

We now construct a linear space by giving its incidence matrix:

```
> R := RMatrixSpace(Integers(), 5, 6);
> I := R![ 1, 0, 1, 1, 0, 0,
>          1, 0, 0, 0, 1, 1,
>          1, 1, 0, 0, 0, 0,
>          0, 1, 1, 0, 1, 0,
>          0, 1, 0, 1, 0, 1];
> L := LinearSpace< 5 | I >;
> L: Maximal;
Linear Space on 5 points with 6 lines
Points: {@ 1, 2, 3, 4, 5 @}
Lines:
    {1, 2, 3},
    {3, 4, 5},
    {1, 4},
    {1, 5},
    {2, 4},
    {2, 5}
```

Finally, we use the minimum weight codewords of the unextended binary Golay code to construct a 4-(23, 7, 1) design. Since we know that this is indeed a 4-design, we set the checking parameter to `false`.

```
> C := GolayCode(GF(2), false);
> C;
[23, 12, 7] Unextended Golay Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 1]
[0 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 0 1 0 1]
[0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 1]
[0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1]
[0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 1]
[0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 1 1 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 1 1]
> minwt := MinimumWeight(C);
> minwt;
7
> wds := Words(C, minwt);
> D := Design< 4, Length(C) | wds : Check := false >;
> D;
```

(continues on next page)

(continued from previous page)

```
4-(23, 7, 1) Design with 253 blocks
```

160.3 The Point-Set and Block-Set of an Incidence Structure

160.3.1 Introduction

An incidence structure created by Magma consists of three objects: the *point-set* P, the *block-set* B, and the incidence structure D itself.

Although called the point-set and block-set, P and B are not actual Magma sets. They simply act as the parent structures for the points and blocks (respectively) of the incidence structure D, enabling easy creation of these objects via the `!` and `.` operators.

The point-set P belongs to the Magma category `IncPtSet`, and the block-set B to the category `IncBlkSet`.

In this section, the functions used to create point-sets, block-sets and the points and blocks themselves are described.

160.3.2 Creating Point-Sets and Block-Sets

As mentioned above, the point-set and block-set are returned as the second and third arguments of any function which creates an incidence structure. They can also be created via the following two functions.

`PointSet(D): Inc -> IncPtSet`

Given an incidence structure D, return the point-set P of D.

`BlockSet(D): Inc -> IncBlkSet`

Given an incidence structure D, return the block-set B of D.

160.3.3 Creating Points and Blocks

For efficiency and clarity, the points and blocks of an incidence structure are given special types in Magma. The category names for points and blocks are `IncPt` and `IncBlk`, respectively. They can be created in the following ways.

`Point(D, i): Inc, RngIntElt -> IncPt`

The i-th point of the incidence structure D.

`P . i: IncPtSet, RngIntElt -> IncPt`

Given the point-set P of an incidence structure D and an integer i, return the i-th point of D.

`Representative(P): IncPtSet -> IncPt`

`Rep(P): IncPtSet -> IncPt`

Given the point-set P of an incidence structure D, return a representative point of D.

`Random(P): IncPtSet -> IncPt`

Given the point-set P of an incidence structure D, return a random point of D.

`P ! x: IncPtSet, Elt -> Incpt`

Given the point-set P of an incidence structure D, return the point of D corresponding to the element x of the indexed set used to create D.

`Block(D, i): Inc, RngIntElt -> IncBlk`

The i-th block of the incidence structure D.

`B . i: IncBlkSet, RngIntElt -> IncBlk`

Given the block-set B of an incidence structure D and an integer i, return the i-th block of D.

Representative(B): IncBlkSet -> IncBlk

Rep(B): IncBlkSet -> IncBlk

Given the block-set B of an incidence structure D, return a representative block of D.

Random(B): IncBlkSet -> IncBlk

Given the block-set B of an incidence structure D, return a random block of D.

B ! S: IncBlkSet, SetEnum -> IncBlk

Given the block-set B of an incidence structure D, and a set S, tries to coerce S into B.

Representative(b): IncBlk -> IncPt

Rep(b): IncBlk -> IncPt

Given a block b of an incidence structure D, return a representative point of D which is incident with b.

Random(b): IncBlk -> IncPt

Given a block b of an incidence structure D, return a random point incident with b.

Example: Points and Blocks of an Incidence Structure (ex-5190d2)

The following example shows how points and blocks of an incidence structure can be created.

```
> V := {@ 2, 4, 6, 8, 10 @};
> D, P, B := IncidenceStructure< V | {2, 4, 6}, {2, 8, 10}, {4, 6, 8} >;
> D;
Incidence Structure on 5 points with 3 blocks
> P;
Point-set of Incidence Structure on 5 points with 3 blocks
> B;
Block-set of Incidence Structure on 5 points with 3 blocks
> B.2;
{2, 8, 10}
> P.4;
8
> P!4;
4
> P.5 eq Point(D, 5);
true
> b := Random(B);
> b;
{2, 4, 6}
> Parent(b);
Block-set of Incidence Structure on 5 points with 3 blocks
> p := Rep(b);
> p;
2
> Parent(p);
Point-set of Incidence Structure on 5 points with 3 blocks
> B!{2, 8, 10};
{2, 8, 10}
```

160.4 General Design Constructions

Each of these functions returns three values:

(i) The incidence structure D;

(ii) The point-set P of D;

(iii) The block-set B of D.

160.4.1 The Construction of Related Structures

All operations defined for incidence structures apply also to near–linear spaces, linear spaces, and designs.

Complement(D): Inc -> Inc

The complement of the incidence structure D.

Dual(D): Inc -> Inc

The dual of the incidence structure D.

Contraction(D, p): Inc, IncPt -> Inc

Given an incidence structure $D = (P, B)$, and a point $p \in P$, form the incidence structure

$$E = (P - \{p\}, \{b - \{p\} : b \in B | p \in b\}).$$

Thus, E is constructed from D by deleting p and retaining only those blocks incident with it.

Contraction(D, b): Inc, IncBlk -> Inc

Given an incidence structure $D = (P, B)$, and a block $b \in B$, form the incidence structure

$$E = (b, \{b \cap c : c \in B | c \neq b\}).$$

Thus, E has point set b and its blocks are the non–empty intersections of b with the blocks of D other than b itself.

Residual(D, b): Inc, IncBlk -> Inc

Given an incidence structure $D = (P, B)$, and a block $b \in B$, form the incidence structure $E = (P - b, B - \{b\})$. Thus, E has point set $P - b$ and its blocks are the non–empty intersections of $P - b$ with the blocks of D.

Residual(D, p): Inc, IncPt -> Inc

Given an incidence structure $D = (P, B)$, and a point $p \in P$, form the incidence structure

$$E = (P - \{p\}, \{x : x \in B | p \notin x\}).$$

Thus, E has point set $P - \{p\}$ and its blocks are the blocks of D which do not contain p.

Simplify(D): Inc -> Inc

Simplify the incidence structure D; i.e., remove repeated blocks from D.

Sum(Q): [Inc] -> Inc

Given a sequence $Q = [D_1, \ldots, D_l]$ of incidence structures, each of which is defined over the same set P of points, form the incidence structure obtained by taking the union of the block sets of $D_1, \ldots, D_l$. Thus, if $D_i = (P, B_i)$ then $D = (P, B_1 \cup \ldots \cup B_l)$.

Union(D, E): Inc, Inc -> Inc

The union of incidence structures D and E. That is, if $D = (P, B)$ and $E = (Q, C)$, then return $U = (P \cup Q, B \cup C)$. The point sets P and Q must be disjoint.

Restriction(D, S): IncNsp, { Incpt } -> IncNsp

The restriction of the (near–)linear space D to the set of points S.

Example: Related Structures of a Design (ex-2db350)

We illustrate some of the above functions with an example.

```
> K := Design< 3, 8 | {1,3,7,8}, {1,2,4,8}, {2,3,5,8}, {3,4,6,8}, {4,5,7,8},
> {1,5,6,8}, {2,6,7,8}, {1,2,3,6}, {1,2,5,7}, {1,3,4,5}, {1,4,6,7}, {2,3,4,7},
> {2,4,5,6}, {3,5,6,7} >;
> CK := Contraction(K, Point(K, 8));
> RK := Residual(K, Block(K, 1));
```

(continues on next page)

(continued from previous page)

```
> K: Maximal;
3-(8, 4, 1) Design with 14 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7, 8 @}
Blocks:
    {1, 3, 7, 8},
    {1, 2, 4, 8},
    {2, 3, 5, 8},
    {3, 4, 6, 8},
    {4, 5, 7, 8},
    {1, 5, 6, 8},
    {2, 6, 7, 8},
    {1, 2, 3, 6},
    {1, 2, 5, 7},
    {1, 3, 4, 5},
    {1, 4, 6, 7},
    {2, 3, 4, 7},
    {2, 4, 5, 6},
    {3, 5, 6, 7}
> CK: Maximal;
2-(7, 3, 1) Design with 7 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7 @}
Blocks:
    {1, 3, 7},
    {1, 2, 4},
    {2, 3, 5},
    {3, 4, 6},
    {4, 5, 7},
    {1, 5, 6},
    {2, 6, 7}
> RK: Maximal;
Incidence Structure on 4 points with 13 blocks
Points: {@ 2, 4, 5, 6 @}
Blocks:
    {2, 4},
    {2, 5},
    {4, 6},
    {4, 5},
    {5, 6},
    {2, 6},
    {2, 6},
    {2, 5},
    {4, 5},
    {4, 6},
    {2, 4},
    {2, 4, 5, 6},
    {5, 6}
> RKS := Simplify(RK);
> RKS: Maximal;
Incidence Structure on 4 points with 7 blocks
Points: {@ 2, 4, 5, 6 @}
Blocks:
    {2, 4},
    {2, 5},
    {4, 6},
    {4, 5},
    {5, 6},
```

(continues on next page)

(continued from previous page)

```
    {2, 6},
    {2, 4, 5, 6}
```

160.4.2 The Witt Designs

The 5–(12, 6, 1) and 5–(24, 8, 1) designs constructed by Witt, also known as the small and large Mathieu designs, respectively, can be constructed in Magma with the following function.

`WittDesign(n): RngIntElt -> Dsgn`

The Witt 5–design on n points, where $n = 12$ or 24.

Example: Contraction of a Witt Design (ex-9c4643)

We construct the Witt 5–(24, 8, 1) design and take its contraction at a point. This contraction is in fact isomorphic to the design constructed above from the unextended binary Golay code.

```
> D, P, B := WittDesign(24);
> D;
5-(24, 8, 1) Design with 759 blocks
> p := P.1;
> Cp := Contraction(D, p);
> Cp;
4-(23, 7, 1) Design with 253 blocks
```

160.4.3 Difference Sets and their Development

Let G be a group of order v and let k and λ be positive integers such that $1 < k < v$. A (v, k, λ) *difference set* for G is a set D of k group elements such that the set

$$\{gh^{-1} : g, h \in D | g \neq h\}$$

contains every non–identity element of G exactly λ times.

`DifferenceSet(p, t): RngIntElt, MonStgElt -> { RngIntResElt }`

The difference set of type given by t (which must be one of `"Q"`, `"H6"`, `"T"`, `"B"`, `"BO"`, `"O"`, `"OO"`, or `"W4"`) corresponding to the prime p. The types have the same interpretation as given by Marshall Hall in [Hal86], pp. 141–142.

`SingerDifferenceSet(n, q): RngIntElt, RngIntElt -> { RngIntResElt }`

The Singer difference set corresponding to a hyperplane of PG(n, q).

`IsDifferenceSet(B): SetEnum -> BoolElt, RngIntElt`

Returns `true` iff B is a difference set over an integer residue class ring or a finite group (with an iterator). If `true`, the value of the parameter λ (i.e., the number of times each non–identity group/ring element appears as a "difference" of elements of B) is also returned.

`Development(B): { RngElt } -> Inc`

Let B be a subset of a magma A which is a difference set relative to A, where A is either the ring Z/mZ, a finite abelian group, or an arbitrary finite group (with an iterator). This function constructs the symmetric design having point set A and whose blocks consist of the sets obtained by translating B by each element of A in turn.

`Development(T): { { Elt } } -> Inc`

Let $T = \{B_1, \ldots, B_l\}$ be a difference family consisting of subsets of a magma A which is either the ring $\mathbb{Z}/m\mathbb{Z}$, a finite abelian group, or an arbitrary finite group (with an iterator). This function constructs the incidence structure with point set A and whose i-th block is the set $\{B_1 \cup \ldots \cup B_l\}$ translated by the i-th element of A.

Example: Developing a Difference Set (ex-b4f959)

The set $\{1, 3, 4, 5, 9\}$, where the elements are residues modulo 11, forms an $(11, 5, 2)$ difference set. We develop this set and construct a 2-$(11, 5, 2)$ design.

```
> Z11 := IntegerRing(11);
> B := { Z11 | 1, 3, 4, 5, 9};
> IsDifferenceSet(B);
true 2
> D := Development(B);
> D: Maximal;
2-(11, 5, 2) Design with 11 blocks
Points: {@ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 @}
Blocks:
    {1, 3, 4, 5, 9},
    {2, 4, 5, 6, 10},
    {0, 3, 5, 6, 7},
    {1, 4, 6, 7, 8},
    {2, 5, 7, 8, 9},
    {3, 6, 8, 9, 10},
    {0, 4, 7, 9, 10},
    {0, 1, 5, 8, 10},
    {0, 1, 2, 6, 9},
    {1, 2, 3, 7, 10},
    {0, 2, 3, 4, 8}
```

We now construct the twin primes (type "T") difference set modulo 323 ($= 17 \times 19$), and its development.

```
> B := DifferenceSet(17, "T");
> D := Development(B);
> D;
2-(323, 161, 80) Design with 323 blocks
```

160.5 Elementary Invariants of an Incidence Structure

All operations defined for incidence structures apply also to near–linear spaces, linear spaces, and designs.

NumberOfPoints(D): Inc -> RngInt

 # P: IncPtSet -> RngIntElt

The cardinality v of the point set P of the incidence structure D.

Points(D): Inc -> { IncPt }

An indexed set E whose elements are the points of the incidence structure D. Note that this creates a standard set and not the point-set of D, in contrast to the function **PointSet**.

Support(D): Inc -> { Elt }

An indexed set E which is the underlying point set of the incidence structure D (i.e., the elements of the set have their "real" types; they are no longer from the category IncPt).

PointDegrees(D): Inc -> [RngIntElt]

A sequence whose i-th term gives the number of blocks containing the i-th point of the design D.

NumberOfBlocks(D): Inc -> RngIntElt

 # B: IncBlkSet -> RngIntElt

The number of blocks b of the incidence structure D with block-set B.

`Blocks(D): Inc -> { IncBlk}`

An indexed set containing the blocks of the incidence structure D. In contrast to the function `BlockSet`, this function returns the collection of blocks of D in the form of a standard set.

`BlockDegrees(D): Inc -> [ RngIntElt ]`

`BlockSizes(D): Inc -> [ RngIntElt ]`

A sequence whose i-th term gives the number of points in the i-th block of the incidence structure D.

`Covalence(D, S): Inc, { IncPt } -> RngIntElt`

Given a subset S of the point set of an incidence structure D, return the number of blocks of D that contain S.

`IncidenceMatrix(D): Inc -> ModMatRngElt`

The incidence matrix of the incidence structure D.

`pRank(D, p): Inc, RngIntElt -> RngIntElt`

The p-rank of the incidence structure D.

160.6 Elementary Invariants of a Design

The following functions can be applied only to designs.

`Parameters(D): Dsgn -> Record`

The parameters t–(v, b, r, k, λ) of the design D returned as a record.

`ReplicationNumber(D): Dsgn -> RngIntElt`

The number of blocks r containing any point of the t–(v, k, λ) design D, where $t > 0$.

`BlockDegree(D): Dsgn -> RngIntElt`

`BlockSize(D): Dsgn -> RngIntElt`

The number of points in a block of the design D.

`Covalence(D, s): Dsgn, RngIntElt -> RngIntElt`

Given a t–(v, k, λ) design D and an integer s such that $0 \leq s \leq t$, return the value of λ_s; i.e., the number of blocks that contain an arbitrary s-subset of the points of D.

`Order(D): Dsgn -> RngIntElt`

The order of the t–(v, k, λ) design D. This is defined only for designs with $t \geq 2$.

`IntersectionNumber(D, i, j): Dsgn, RngIntElt, RngIntElt -> RngIntElt`

The block intersection number λ_i^j; i.e., the number of blocks of the design D containing an i-set and disjoint from a j-set. The arguments i and j must satisfy $i + j \leq t$.

`PascalTriangle(D): Dsgn -> SeqEnum`

The "Pascal triangle" of the design D, returned as a sequence; the i-th element of the sequence is a sequence representing the i-th row of the triangle. That is, the i-th element of the sequence is

$$[\lambda_0^{i-1}, \lambda_1^{i-2}, \ldots, \lambda_{i-1}^0].$$

If D is a Steiner t–design, then the triangle returned has $k + 1$ rows (where k is the size of a block of D); otherwise the triangle has $t + 1$ rows.

Example: Invariants of a Fano Plane Design (ex-3a71cb)

We illustrate some of the functions of the previous two sections with an example.

```
> F := Design< 2, 7 | {1,2,4}, {1,3,7}, {2,3,5}, {1,5,6}, {3,4,6}, {4,5,7},
>   {2,6,7} >;
> G := IncidenceStructure< 7 | Blocks(F), {1, 3, 7}, {1, 2, 4},
>   {3, 4, 5}, {2, 3, 6}, {2, 5, 7}, {1, 5, 6}, {4, 6, 7} >;
> F;
2-(7, 3, 1) Design with 7 blocks
> G;
Incidence Structure on 7 points with 14 blocks
> Points(G);
{@ 1, 2, 3, 4, 5, 6, 7 @}
> Blocks(F);
{@
    {1, 2, 4},
    {1, 3, 7},
    {2, 3, 5},
    {1, 5, 6},
    {3, 4, 6},
    {4, 5, 7},
    {2, 6, 7}
@}
> IncidenceMatrix(F);
[1 1 0 1 0 0 0]
[1 0 1 0 0 0 1]
[0 1 1 0 1 0 0]
[1 0 0 0 1 1 0]
[0 0 1 1 0 1 0]
[0 0 0 1 1 0 1]
[0 1 0 0 0 1 1]
> P := Points(F);
> P, Universe(P);
{@ 1, 2, 3, 4, 5, 6, 7 @}
Point-set of 2-(7, 3, 1) Design with 7 blocks
> S := Support(F);
> S, Universe(S);
{@ 1, 2, 3, 4, 5, 6, 7 @}
Integer Ring
> Covalence(G, {1, 2});
2
> Order(F);
2
> PascalTriangle(F);
        7
      4   3
    2   2   1
  0   2   0   1
```

160.7 Operations on Points and Blocks

In incidence structures, particularly simple ones, blocks are basically sets. For this reason, the elementary set operations such as `join`, `meet` and `subset` have been made to work on blocks. However, blocks are not true Magma enumerated sets, and so the functions `Set` and `Support` below have been provided to convert a block to an enumerated set of points for other uses.

`p in B: IncPt, IncBlk -> BoolElt`

Returns `true` if point p lies in block B, otherwise `false`.

`p notin B: IncPt, IncBlk -> BoolElt`

Returns `true` if point p does not lie in block B, otherwise `false`.

`S subset B: { IncPt }, IncBlk -> BoolElt`

Given a subset S of the point set of the incidence structure D and a block B of D, return `true` if the subset S of points lies in B, otherwise `false`.

`S notsubset B: { IncPt }, IncBlk -> BoolElt`

Given a subset S of the point set of the incidence structure D and a block B of D, return `true` if the subset S of points does not lie in B, otherwise `false`.

`PointDegree(D, p): Inc, IncPt -> RngIntElt`

The number of blocks of the incidence structure D that contain the point p.

`BlockDegree(D, B): Inc, IncBlk -> RngIntElt`

`BlockSize(D, B): Inc, IncBlk -> RngIntElt`

`# B: IncBlk -> RngIntElt`

The number of points contained in the block B of the incidence structure D.

`Set(B): IncBlk -> { IncPt }`

The set of points contained in the block B.

`Support(B): IncBlk -> { Elt }`

The set of underlying points contained in the block B (i.e., the elements of the set have their "real" types; they are no longer from the category IncPt).

`IsBlock(D, S): Inc, IncBlk -> BoolElt, IncBlk`

`IsBlock(D, S): Inc, SetEnum -> BoolElt, IncBlk`

Returns `true` iff the set (or block) S represents a block of the incidence structure D. If `true`, also returns one such block.

`Line(D, p, q): Inc, IncPt, IncPt -> IncBlk`

`Block(D, p, q): Inc, IncPt, IncPt -> IncBlk`

A block of the incidence structure D containing the points p and q (if one exists). In linear spaces, such a block exists and is unique (assuming p and q are different).

`ConnectionNumber(D, p, B): Inc, IncPt, IncBlk -> RngIntElt`

The connection number $c(p, B)$; i.e., the number of blocks joining p to B in the incidence structure D.

Example: Operations on Points and Blocks (ex-f06ff8)

The following examples uses some of the functions of the previous section.

```
> D, P, B := Design< 2, 7 | {3, 5, 6, 7}, {2, 4, 5, 6}, {1, 4, 6, 7},
>   {2, 3, 4, 7}, {1, 2, 5, 7}, {1, 2, 3, 6}, {1, 3, 4, 5} >;
> D: Maximal;
2-(7, 4, 2) Design with 7 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7 @}
Blocks:
    {3, 5, 6, 7},
    {2, 4, 5, 6},
    {1, 4, 6, 7},
    {2, 3, 4, 7},
    {1, 2, 5, 7},
    {1, 2, 3, 6},
    {1, 3, 4, 5}
> P.1 in B.1;
false
> P.1 in B.3;
```

(continues on next page)

(continued from previous page)

```
true
> {P| 1, 2} subset B.5;
true
> Block(D, P.1, P.2);
{1, 2, 5, 7}
> b := B.4;
> b;
{2, 3, 4, 7}
> b meet {2, 8};
{ 2 }
> S := Set(b);
> S, Universe(S);
{ 2, 3, 4, 7 }
Point-set of 2-(7, 4, 2) Design with 7 blocks
> Supp := Support(b);
> Supp, Universe(Supp);
{ 2, 3, 4, 7 }
Integer Ring
```

160.8 Elementary Properties of Incidence Structures and Designs

Testing the t-balance: A general Note on Parameter `Al`

A parameter `Al` is provided to allow users to specify the algorithm to be used for balance testing. The default value is `"NoOrbits"`, which applies a "brute force" test. One alternative is `"Orbits"`, which uses the orbits of t-sets under the automorphism group of the incidence structure under consideration. This is much faster than `"NoOrbits"` for some cases, but slower for others.

The other alternative for checking if an object is t-balanced is `"FastBalanceTest"`: It is also a "brute force" test but whose implementation allows for a dramatic improvement in efficiency, especially for larger t ($t \geq 4$). This implementation has a drawback however as it may necessitate more memory space than is available, thus resulting in an out-of-memory error. It is for this reason that `"FastBalanceTest"` is not the default setting for `Al`, but as a general rule we strongly recommend its usage as we surmise that execution should successfully complete in most cases.

`IsSimple(D): Inc -> BoolElt`

Returns `true` if and only if the incidence structure D has no repeated blocks.

`IsTrivial(D): Inc -> BoolElt`

Returns `true` if and only if the incidence structure D is a trivial incidence structure.

`IsSelfDual(D): Inc -> BoolElt`

Returns `true` if and only if the incidence structure D is self-dual, that is, if D is isomorphic to its dual.

`IsUniform(D): Inc -> BoolElt, RngIntElt`

Returns `true` if and only if the incidence structure D is uniform; that is, each block contains the same number of points. If `true`, also returns the blocksize.

`IsNearLinearSpace(D): Inc -> BoolElt`

Returns `true` if and only if the incidence structure D is a near–linear space.

`IsLinearSpace(D): Inc -> BoolElt`

Returns `true` if and only if the incidence structure D is a linear space.

`IsDesign(D, t: parameters): Inc, RngIntElt -> BoolElt, RngIntElt`

```
Al: MonStgElt                    Default: "NoOrbits"
```

Returns `true` if and only if the incidence structure D is a t–design. If `true`, then the number of blocks of D containing a general t-set is also returned. The optional parameter `Al` can be used to specify the algorithm used for balance testing, see the introduction to Section *Elementary Properties of Incidence Structures and Designs* for a full description of its usage.

`IsBalanced(D, t: parameters): Inc, RngIntElt -> BoolElt, RngIntElt`

```
Al: MonStgElt                    Default: "NoOrbits"
```

Returns `true` if and only if the incidence structure D is balanced (with respect to t). If `true`, then the number of blocks of D containing a general t-set is also returned. The optional parameter `Al` can be used to specify the algorithm used for balance testing, see the introduction to Section *Elementary Properties of Incidence Structures and Designs* for a full description of its usage.

`IsComplete(D): Inc -> BoolElt`

Return `true` if and only if D is the complete design.

`IsSymmetric(D): Dsgn -> BoolElt`

Return `true` if and only if the design D is symmetric.

`IsSteiner(D, t): Dsgn, RngIntElt -> BoolElt`

```
Al: MonStg                    Default: "NoOrbits"
```

Return `true` if and only if the design D is a Steiner t–design. The optional parameter `Al` can be used to specify the algorithm used for balance testing, see the introduction to Section *Elementary Properties of Incidence Structures and Designs* for a full description of its usage.

`IsPointRegular(D): IncNsp -> BoolElt, RngIntElt`

Return `true` if and only if the (near–)linear space D is point regular. If `true`, the point regularity is also returned.

`IsLineRegular(D): IncNsp -> BoolElt, RngIntElt`

Return `true` if and only if the (near–)linear space D is line regular. If `true`, the line regularity is also returned.

160.9 Resolutions, Parallelisms, and Parallel Classes

Let D be an incidence structure with v points. A *resolution* of D is a partition of the blocks of D into classes C_i such that each class is a 1-design with v points and index λ. The positive integer λ is called the *index* of the resolution. A resolution with $\lambda = 1$ is called a *parallelism*. In this case the classes C_i are called *parallel classes*.

All the functions which deal with resolutions apply to general incidence structures. The functions `HasParallelism`, `AllParallelisms`, `HasParallelClass`, `IsParallelClass`, and `AllParallelClasses` require that the incidence structure be uniform. The function `IsParallelism` however applies to a general incidence structure.

`HasResolution(D): Inc -> BoolElt, { SetEnum}, RngIntElt`

Return `true` if and only if the incidence structure D has a resolution. If `true`, one resolution is returned as the second value of the function and the index of the resolution is returned as the third value.

`HasResolution(D, λ): Inc, RngIntElt -> BoolElt, { SetEnum}`

Return `true` if and only if the incidence structure D has a resolution with index λ. If `true`, one such resolution is returned as the second value of the function.

`AllResolutions(D): Inc -> SeqEnum`

Returns all resolutions of the incidence structure D.

AllResolutions(D, λ): Inc, RngIntElt -> SeqEnum

Returns all resolutions of the incidence structure D having index λ. When the problem is to find all parallelisms ($\lambda = 1$) in a uniform design D, it is best to use the function AllParallelisms described below.

IsResolution(D, P): Inc, SetEnum[SetEnum] -> BoolElt, RngIntElt

Returns true if and only if the set P of blocks (or sets) is a resolution of the incidence structure D. If true, also returns the index of the resolution as the second value of the function.

HasParallelism(D: parameters): Inc, RngIntElt -> BoolElt, { SetEnum}

Al: MonStg	Default: "Backtrack"

Returns true if and only if the uniform incidence structure D has a parallelism. If true, a parallelism is returned as the second value of the function. The parameter Al allows the user to specify the algorithm used.

Al := "Backtrack": A backtrack search is employed. This is in general very efficient when the design D is parallelizable, irrespective of the number of blocks. In particular, it compares very favourably with the algorithm Clique, described below. The later algorithm is recommended when the design appears to have no parallelism. Backtrack is the default.

Al := "Clique": This algorithm proceeds in two stages. Assume that D has v points and b blocks of size k. Define the graph G_1 of D to be the graph whose vertices are the blocks of D and where two blocks are adjacent if and only if they are disjoint. The first stage of the Clique algorithm involves constructing the graph G_1 and finding all cliques of size v/k in G_1. Any such clique is a parallel class of D. The second stage involved building a graph G_2 of D where the vertices are the parallel classes of D and two parallel classes are adjacent if and only if they are disjoint. The second stage of the Clique algorithm involves searching for cliques of size $b/(v/k)$ in G_2. If such a clique exists then it yields a parallelism of D. (This algorithm was communicated by Vladimir Tonchev).

The clique algorithm is recommended when the design is suspected of having no parallelism. As an example, in the case of a non–parallelizable design with 272 blocks, Clique took around 0.1 second to complete as compared to 2 seconds for the backtrack algorithm. For a non–parallelizable design having 6642 blocks, Clique took around four hours to complete as compared to Backtrack which did not complete in 15 hours. Apart from performing very poorly when D has a parallelism, Clique may require a considerable amount of memory. It is recommended that Backtrack be tried first, and if it does not complete in a reasonable amount of time, then Clique should be used.

AllParallelisms(D): Inc -> SeqEnum

Returns all parallelisms of the uniform incidence structure D. This function is to be preferred to the function AllResolutions(D, 1) when D is uniform. This is the case since the implementation of AllParallelClasses implies finding cliques in graphs (see the algorithm Clique described in the function HasParallelism), while AllResolutions performs a full backtrack search.

IsParallelism(D, P): Inc, SetEnum[SetEnum] -> BoolElt, RngIntElt

Returns true if and only if the set P of blocks (or sets) is a parallelism of the incidence structure D.

HasParallelClass(D): Inc -> BoolElt, { IncBlk}

Returns true if and only if the uniform incidence structure D has a parallel class.

IsParallelClass(D, B, C): Inc, IncBlk, IncBlk -> BoolElt, { IncBlk}

Returns true if and only if the uniform incidence structure D has a parallel class containing the blocks B and C. If such a parallel class does exist, one is returned as the second value of the function.

AllParallelClasses(D): Inc -> SeqEnum

Returns all parallel classes of the uniform incidence structure D.

Example: Resolutions and Parallel Classes (ex-102d24)

Some of the above functions are illustrated here.

```
> D := IncidenceStructure< 6 | {1,2,3}, {4,5,6}, {1,3,4}, {2,5,6}>;
> bool, R, lambda := HasResolution(D);
> bool;
true
> R;
{
  {
    {1, 2, 3},
    {4, 5, 6},
    {1, 3, 4},
    {2, 5, 6}
  }
}
> lambda;
2
> HasResolution(D,2);
true {
  {
    {1, 2, 3},
    {4, 5, 6},
    {1, 3, 4},
    {2, 5, 6}
  }
}
> AllResolutions(D);
[
  {
    {
      {1, 2, 3},
      {4, 5, 6},
      {1, 3, 4},
      {2, 5, 6}
    }
  },
  {
    {
      {1, 2, 3},
      {4, 5, 6}
    },
    {
      {1, 3, 4},
      {2, 5, 6}
    }
  }
]
> HasParallelism(D);
true {
  {
    {1, 2, 3},
    {4, 5, 6}
  },
  {
    {1, 3, 4},
    {2, 5, 6}
```

(continues on next page)

(continued from previous page)

```
  }
}
> V := PointSet(D);
> S := { PowerSet(PowerSet(V)) |
>        { {1, 2, 3}, {4, 5, 6} }, { {1, 3, 4}, {2, 5, 6} } };
> IsParallelism(D, S);
true
> B := BlockSet(D);
> S := { { B.1, B.2 }, {B.3, B.4 }};
> IsParallelism(D, S);
true
> AllParallelClasses(D);
{
  {
    {1, 2, 3},
    {4, 5, 6}
  },
  {
    {1, 3, 4},
    {2, 5, 6}
  }
}
```

160.10 Conversion Functions

IncidenceStructure(I): Inc -> Inc

Given an incidence structure I (of any type), return the same structure as a "true" incidence structure (i.e., belonging to the category Inc).

NearLinearSpace(I): Inc -> IncNsp

Given an incidence structure I (of any type), return the same structure as a near–linear space (i.e., belonging to the category IncNsp), if possible.

LinearSpace(I): Inc -> IncLsp

Given an incidence structure I (of any type), return the same structure as a linear space (i.e., belonging to the category IncLsp), if possible.

Design(I, t): Inc, RngIntElt -> Dsgn

Given an incidence structure I (of any type), return the same structure as a t–design (i.e., belonging to the category Dsgn), if possible.

Example: Converting Incidence Structures (ex-97a8b8)

The following example illustrates some of the functions of the previous two sections.

```
> I := IncidenceStructure< 8 | {1, 2, 3, 4}, {1, 3, 5, 6}, {1, 4, 7, 8},
>   {2, 5, 6, 7}, {2, 5, 7, 8}, {3, 4, 6, 8} >;
> IsDesign(I, 1);
true 3
> IsDesign(I, 2);
false
> D := Design(I, 1);
> D;
1-(8, 4, 3) Design with 6 blocks
> IsSteiner(D, 1);
false
```

(continues on next page)

(continued from previous page)

```
> IsNearLinearSpace(I);
false
```

160.11 Identity and Isomorphism

Incidence structures $D_1 = (P_1, B_1, I_1)$ and $D_2 = (P_2, B_2, I_2)$ are *identical* if $P_1 = P_2$, $B_1 = B_2$, and $I_1 = I_2$.

`D eq E: Inc, Inc -> BoolElt`

Return `true` if the incidence structures D and E are identical, otherwise `false`.

`D ne E: Inc, Inc -> BoolElt`

Return `true` if the incidence structures D and E are not identical, otherwise `false`.

`IsIsomorphic(D, E: parameters): Inc, Inc -> BoolElt, Map`

```
AutomorphismGroups: MonStgElt                    Default: "None"
```

Return `true` if the incidence structures D and E are isomorphic, otherwise `false`. If they are isomorphic, then an isomorphism is returned as the second value. The function first computes none, one, or both of the automorphism groups of the left and right incidence structures. In difficult examples, this may significantly speed up testing for isomorphism. The parameter `AutomorphismGroups`, with valid string values `"Both"`, `"Left"`, `"Right"`, `"None"`, may be used to specify which of the automorphism groups should be constructed if not already known. The default is `"None"`.

160.12 The Automorphism Group of an Incidence Structure

The automorphism group A of an incidence structure D is always presented as a permutation group G acting on the standard support. The reasons for this include the fact that the support chosen when computing the automorphism group is often quite complicated. Further, if the group is represented as acting on a set of objects relating to the incidence structure, printed permutations are often unreadable. The support used when constructing the automorphism group of an incidence structure D is either the point set of D (when D is simple), or the disjoint union of the point set with the block set (when D has repeated blocks).

The automorphism group G of D does not act directly on D. Instead, G-sets are used to transfer the action of G to various sets associated with D. The two most important G-sets, corresponding to action of G on the point set and on the block set, are returned by each of the functions provided for constructing automorphism group or some specified subgroup of it.

In some circumstances, rather than viewing automorphisms as group elements, it is desirable to view them is as mappings of D into itself. Associated with each incidence structure is a mapping structure, $\mathrm{Aut}(D)$, which denotes the set of automorphisms of D. Note that $\mathrm{Aut}(D)$ is the *parent* of the automorphisms of G so that the function `Aut(D)` simply creates a shell structure rather than the actual automorphism group of D. A transfer map is provided to convert a permutation of the automorphism group G into a mapping belonging to $Aut(D)$.

160.12.1 Construction of Automorphism Groups

`AutomorphismGroup(D): Inc -> GrpPerm, GSet, GSet, PowMap, Map`

Construct the automorphism group G of the incidence structure D. The set on which G acts depends upon whether or not the design is simple. Suppose D has v points and b blocks. If the incidence structure D is simple, the automorphism group is constructed in its action on the points of D. It is returned acting on the standard support $\{1, \ldots, \mathrm{v}\}$, where i corresponds to the i-th point of D. If D is not simple, it is constructed in its action of the disjoint union of the point set and block set of D. Again, it is returned acting on the standard support $\{1, \ldots, \mathrm{v+b}\}$, where $1 \leq i \leq v$

corresponds to the i-th point of D and $v+1 \leq i \leq v+b$ corresponds to the $(i-v)$-th point of D. The function returns:

(i) The automorphism group G of D as described above;

(ii) A G-set Y corresponding to the action of G on the point set of D.

(iii) A G-set W corresponding to the action of G on the block set of D.

(iv) The *Aut* structure S for D; and

(v) A transfer map $t : G \to S$.

Given a permutation g of G, $t(g)$ is the *mapping* of D into itself that corresponds to the automorphism group element g. The G-sets Y and W should be used whenever it is necessary to have G act on the points or lines of D.

`AutomorphismSubgroup(D): Inc -> GrpPerm, PowMap, Map`

A cyclic subgroup H of the automorphism group G of the incidence structure D. The purpose of this function is to terminate the search for automorphisms of D as soon as a non–trivial automorphism is found. The function returns:

(i) The automorphism group H of D as described above;

(ii) The *Aut* structure S for D; and

(iii) A transfer map $t : G \to S$.

Given a permutation g of H, $t(g)$ is the *mapping* of D into itself that corresponds to the automorphism group element g.

`AutomorphismGroupStabilizer(D, k): Inc, RngIntElt -> GrpPerm, PowMap, Map`

The subgroup H of the automorphism group G of the incidence structure D, which stabilizes the first k base points of G. This function is provided so as to return a subgroup of G that is sometimes easier to compute than all of G. The function returns:

(i) The automorphism group H of D as described above;

(ii) The *Aut* structure S for D; and

(iii) A transfer map $t : G \to S$.

Given a permutation g of H, $t(g)$ is the *mapping* of D into itself that corresponds to the automorphism group element g.

`PointGroup(D): Inc -> GrpPerm, GSet`

The automorphism group of the incidence structure D given in its action on the point set of D, together with the points G-set.

`BlockGroup(D): Inc -> GrpPerm`

The automorphism group of the incidence structure D given in its action on the block set of D.

`Aut(D): Inc -> PowMapAut, Map`

The power structure A of all automorphisms of the incidence structure D, together with the transfer map $t : Sym(n) \to A$, where the points of $Sym(n)$ are assumed to be in one–to–one correspondence with the natural support for the automorphism group of D.

Example: Automorphism Group of a Hadamard Design `(ex-d8bb15)`

We construct a $3-(16,8,3)$ Hadamard design and investigate its automorphism group. The first step is to construct a Hadamard matrix of order 16.

```
> R := MatrixRing(Integers(), 4);
> H := R ! [1,1,1,-1, 1,1,-1,1, 1,-1,1,1, -1,1,1,1];
> L := TensorProduct(H, -H);
> D, P, B := HadamardRowDesign(L, 1);
> D;
3-(16, 8, 3) Design with 30 blocks
> G, pg, bg, A, t := AutomorphismGroup(D);
> G;
Permutation group G acting on a set of cardinality 16
Order = 322560 = 2^10 * 3^2 * 5 * 7
    (9, 15)(10, 16)(11, 13)(12, 14)
    (5, 11, 7, 9)(6, 12, 8, 10)(13, 15)(14, 16)
    (5, 9, 14)(6, 10, 13)(7, 11, 16)(8, 12, 15)
    (1, 2)(7, 8)(9, 13, 10, 14)(11, 16, 12, 15)
    (2, 5)(4, 7)(10, 13)(12, 15)
    (3, 9)(4, 10)(7, 13)(8, 14)
    (3, 8, 15, 12)(4, 7, 16, 11)(5, 9)(6, 10)
    (5, 6)(7, 8)(13, 14)(15, 16)
    (3, 12, 4, 11)(7, 16, 8, 15)(9, 10)(13, 14)
    (3, 7)(4, 8)(11, 15)(12, 16)
    (3, 8)(4, 7)(9, 10)(11, 15)(12, 16)(13, 14)
    (9, 10)(11, 12)(13, 14)(15, 16)
    (9, 13)(10, 14)(11, 15)(12, 16)
> CompositionFactors(G);
    G
    |  Alternating(8)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
```

160.12.2 Action of Automorphisms

As noted at the beginning of the section, the automorphism group G of an incidence structure D is given in its action on the standard support and it does not act directly on D. The action of G on D is obtained using the G-set mechanism. The two basic G-sets associated with D correspond to the action of G on the set of points P and the set of blocks B of D. These two G-sets are given as return values of the function `AutomorphismGroup` or may be constructed directly. Additional G-sets associated with D may be built using the G-set constructors. Given a G-set Y for G, the action of G on Y may be studied using the permutation group functions that allow a G-set as an argument. In this section, only a few of the available functions are described: see the section on G-sets for a complete list.

`Image(g, Y, y): GrpPermElt, GSet, Elt -> Elt`

Let G be a subgroup of the automorphism group for the incidence structure D and let Y be a G-set for G. Given an element y belonging either to Y or to a G-set derived from Y, find the image of y under G.

`Orbit(G, Y, y): GrpPerm, GSet, Elt -> GSet`

Let G be a subgroup of the automorphism group for the incidence structure D and let Y be a G-set for G. Given an element y belonging either to Y or to a G-set derived from Y, construct the orbit of y under G.

`Orbits(G, Y): GrpPerm, GSet -> [ GSet ]`

Let G be a subgroup of the automorphism group for the incidence structure D and let Y be a G-set for G. This function constructs the orbits of the action of G on Y.

`Stabilizer(G, Y, y): GrpPerm, GSet, Elt -> GrpPerm`

Let G be a subgroup of the automorphism group for the incidence structure D and let Y be a G-set for G. Given an element y belonging either to Y or to a G-set derived from Y, construct the stabilizer of y in G.

`Action(G, Y): GrpPerm, GSet -> Hom(Grp), GrpPerm, GrpPerm`

Given a subgroup G of the automorphism group of the incidence structure D, and a G-set Y for G, construct the homomorphism $\phi : G \to L$, where the permutation group L gives the action of G on the set Y. The function returns:

(a) The natural homomorphism $\phi : G \to L$;

(b) The induced group L;

(c) The kernel of the action (a subgroup of G).

`ActionImage(G, Y): GrpPerm, GSet -> GrpPerm`

Given a subgroup G of the automorphism group of the incidence structure D, and a G-set Y for G, construct the permutation group L giving the action of G on the set Y.

`ActionKernel(G, Y): GrpPerm, GSet -> GrpPerm`

Given a subgroup G of the automorphism group of the incidence structure D, and a G-set Y for G, construct the kernel of the action of G on the set Y.

`IsPointTransitive(D): Inc -> BoolElt`

Returns `true` iff the automorphism group of the incidence structure D acts transitively on the point set of D.

`IsBlockTransitive(D): Inc -> BoolElt`

Returns `true` iff the automorphism group of the incidence structure D acts transitively on the block set of D.

Example: Automorphism Group of a Witt Design (ex-f58989)

In the following example, the automorphism group of the Witt design on 12 points is constructed.

```
> D, P, B := WittDesign(12);
> A, PY, BY := AutomorphismGroup(D);
> A;
Permutation group Aut acting on a set of cardinality 12
Order = 95040 = 2^6 * 3^3 * 5 * 11
    (1, 2)(5, 8)(6, 11)(10, 12)
    (2, 3)(5, 12)(6, 8)(10, 11)
    (3, 4)(5, 10)(6, 11)(8, 12)
    (4, 7)(5, 8)(6, 12)(10, 11)
    (5, 11, 12, 6)(7, 8, 9, 10)
    (5, 9, 12, 7)(6, 10, 11, 8)
    (5, 12)(6, 11)(7, 9)(8, 10)
> a := A!(1, 2, 3, 4)(5, 6, 12, 11);
> 4^a;
1
> {1, 2, 3}^a;
{ 2, 3, 4 }
> Image(a, BY, Block(D, 3));
{1, 2, 3, 8, 11, 12}
> S2 := SylowSubgroup(A, 2);
```

(continues on next page)

(continued from previous page)

```
> S2;
Permutation group S2 acting on a set of cardinality 12
Order = 64 = 2^6
   (1, 11, 9, 4)(6, 10, 12, 7)
   (1, 9)(2, 12)(3, 7)(4, 11)(5, 10)(6, 8)
   (1, 4)(3, 5)(7, 10)(9, 11)
> Stabilizer(S2, BY, Block(D, 4));
Permutation group acting on a set of cardinality 12
Order = 2
    (1, 9)(3, 5)(6, 7)(10, 12)
> Stabilizer(S2, 6);
Permutation group acting on a set of cardinality 12
Order = 8 = 2^3
    (1, 4)(3, 5)(7, 10)(9, 11)
    (1, 4, 9, 11)(2, 5, 8, 3)
> 3^S2;
GSet{ 2, 3, 5, 6, 7, 8, 10, 12 }
> IsPointTransitive(D);
true
> IsBlockTransitive(D);
true
```

We calculate the action of the automorphism group of the design on its blocks.

```
> H := ActionImage(A, BY);
> #H;
95040
> H;
Permutation group H acting on a set of cardinality 132
Order = 95040 = 2^6 * 3^3 * 5 * 11
```

160.13 Incidence Structures, Graphs, and Codes

`IncidenceStructure(G): Grph -> Inc`

Construct the incidence structure D corresponding to the graph G, where the blocks of D correspond to the edges of G.

`PointGraph(D): Inc -> Grph`

The point graph G of the incidence structure D. The graph G has the same point set as D and two vertices u and v of G are adjacent whenever there is a block of D containing both u and v.

`BlockGraph(D): Inc -> Grph`

The block graph of the incidence structure D, i.e., the point graph of the dual of D.

`IncidenceGraph(D): Inc -> Grph`

The incidence graph of the incidence structure D. This bipartite graph has as vertex set the union of the point set P and block set B of D. A vertex $p \in P$ is adjacent to a vertex $b \in B$ whenever $p \in b$.

`LinearCode(D, K): Inc, FldFin -> Code`

Given an incidence structure D with v points and a finite field K, this function returns the linear code C of length v generated by the characteristic functions of the blocks of D considered as vectors of the K-space $K^{(v)}$.

Example: Code of the Witt Design `(ex-09b77a)`

The linear code of the Witt 5-(24, 8, 1) design over GF(2) is the extended Golay code over GF(2).

```
> D := WittDesign(24);
> C := LinearCode(D, GF(2));
> C eq GolayCode(GF(2), true);
true
```

160.14 Automorphisms of Matrices

Matrices may be regarded as defining a design with the entry of a matrix in a particular row and column defining the incidence of the row and column. The automorphism group of a matrix is the set of permutations of the rows and columns of the matrix that leave the matrix unchanged.

`M ^ x: Mtrx, GrpPermElt -> Mtrx`

The action of a permutation on a matrix by permuting rows and columns. If M has r rows and c columns then x must have degree $r + c$ and fix the set $R = \{1, \ldots, r\}$. The action of x on R gives the permutation of the rows of the matrix. The remainder of x gives the action on columns.

`AutomorphismGroup``(M : parameters): Mtrx -> GrpPerm`

```
RowColours: SeqEnum[RngIntElt]                  Default:
ColColours: SeqEnum[RngIntElt]                  Default:
Subgroup  : GrpPerm                             Default:
Al        : MonStgElt                           Default: "Default"
```

Computes the group of all permutations x such that $M^x = M$. The method used is a backtrack search as described by J. Leon.

If M has r rows then `RowColours` may be set to a sequence of integers with length r. This defines a partition of the rows of M where a block is the set of rows with the same number in the sequence. The group found in this case will be the subgroup of the automorphism group of M that fixes this row partition. The `ColColours` parameter may be used to partition the columns of the matrix.

The `Subgroup` parameter may be set to a known subgroup of the automorphism group. This may speed the backtrack search. The default value is the trivial subgroup.

The parameter `Al` is kept for backward compatibility.

`IsIsomorphic``(M, N: parameters): Mtrx, Mtrx -> BoolElt, GrpPermElt`

```
LeftRowColours : SeqEnum[RngIntElt]                 Default:
LeftColColours : SeqEnum[RngIntElt]                 Default:
RightRowColours: SeqEnum[RngIntElt]                 Default:
RightColColours: SeqEnum[RngIntElt]                 Default:
```

Finds a permutation x such that $M^x = N$, if such exists. If a permutation is found return values are `true` and the permutation, otherwise returns `false`. The algorithm used is a backtrack search as for `AutomorphismGroup` above.

Parameters

may be used to describe partitions of the rows and columns of M (`Left`) and of N (`Right`) that are to be preserved by x.

The user may also supply subgroups of the automorphism groups of M (as `LeftSubgroup`) and of N (as `RightSubgroup`), if such subgroups are known. These may speed the course of the algorithm.

Example: Automorphisms of a Fano Polytope Matrix `(ex-5ad3a5)`

We construct a matrix from a Fano polytope and compute its automorphism group.

```
> M := VertexFacetHeightMatrix(PolytopeSmoothFanoDim3(10)); M;
[1 0 0 0 2 1 2 2]
[0 0 0 1 1 2 2 2]
[0 0 1 2 0 2 2 1]
[2 0 1 0 2 0 2 1]
[0 2 0 1 1 2 0 2]
[1 2 0 0 2 1 0 2]
[0 2 1 2 0 2 0 1]
[2 2 1 0 2 0 0 1]
[2 2 2 1 1 0 0 0]
[1 2 2 2 0 1 0 0]
[1 0 2 2 0 1 2 0]
[2 0 2 1 1 0 2 0]
> A := AutomorphismGroup(M); A;
Permutation group A acting on a set of cardinality 20
Order = 24 = 2^3 * 3
  (2, 4)(3, 12)(5, 8)(7, 9)(13, 18)(15, 16)(17, 20)
  (1, 5)(2, 6)(3, 8)(4, 7)(9, 11)(10, 12)(13, 16)(14, 19)(17, 18)
  (1, 2)(3, 4)(5, 6)(7, 8)(9, 10)(11, 12)(13, 16)(17, 18)
> Nrows(M), Ncols(M);
12, 8
> Orbits(A);
[
  GSet{@ 14, 19 @},
  GSet{@ 13, 18, 16, 17, 15, 20 @},
  GSet{@ 1, 5, 2, 8, 6, 4, 3, 7, 12, 9, 10, 11 @}
]
```

The automorphism group is transitive on the rows of the matrix. Now dualize the matrix and test for isomorphism.

```
> D := Matrix(12, 8, [2-x:x in Eltseq(M)]);
> f, x := IsIsomorphic(M, D); f, x;
true (2, 4)(3, 12)(5, 8)(7, 9)(14, 19)(15, 17)(16, 20)
> M^x eq D;
true
```

160.15 Bibliography

- [Hal86] Marshall Hall. *Combinatorial Theory*. New York: Wiley, 2nd edition, 1986.

Chapter 161

Hadamard Matrices

161.1 Introduction

A *Hadamard matrix* is an $n \times n$ matrix, all of whose entries are ± 1, such that every pair of rows and every pair of columns differ in exactly $\frac{n}{2}$ places. Two such matrices are considered equivalent if one can be transformed into the other by performing row swaps, column swaps, row negations, or column negations. The problem of deciding whether two Hadamard matrices are equivalent is hard.

Magma contains several specialised routines for working with Hadamard matrices; of special note is the introduction of a canonical form for such matrices (based on Brendan McKay's `nauty` program or Adolfo Piperno's Traces package). This leads to a much faster equivalence algorithm than previously available.

161.2 Equivalence Testing

`IsHadamard(H): AlgMatElt -> BoolElt`

Returns `true` if and only if the matrix H is a Hadamard matrix.

`HadamardNormalize(H): AlgMatElt -> AlgMatElt`

Given a Hadamard matrix H, returns a normalized matrix equivalent to H. This matrix is created by negating rows and columns to ensure that the first row and column consist entirely of ones.

`HadamardCanonicalForm(H : parameters): AlgMatElt -> AlgMatElt, AlgMatElt, AlgMatElt`

```
Al: MonStgElt                    Default: "Traces"
```

Given a Hadamard matrix H, returns a Hadamard-equivalent matrix H' and transformation matrices X and Y such that $H' = XHY$. The parameter `Al` may be set to `"nauty"` or `"Traces"` (the default). The matrix H' is canonical in the sense that all matrices that are Hadamard-equivalent to H (and no others) will produce the same matrix H' with the given algorithm. Note that the canonical form produced by nauty may be different from that produced by Traces.

`HadamardInvariant(H): AlgMatElt -> [ RngIntElt ]`

Returns a sequence S of integers giving the 4-profile of the Hadamard matrix H. All Hadamard-equivalent matrices have the same 4-profile, but so may some inequivalent ones. Thus this test may determine inequivalence of two matrices more cheaply than performing a full equivalence test, but cannot establish equivalence.

`IsHadamardEquivalent(H, J : parameters): AlgMatElt, AlgMatElt -> BoolElt, AlgMatElt, AlgMatElt`

```
Al: MonStgElt                    Default: "Traces"
```

Returns `true` if and only if the Hadamard matrices H and J are Hadamard equivalent. The parameter `Al` may be set to `"Leon"`, `"nauty"`, or `"Traces"` (the default). If the `"nauty"` or `"Traces"` options are chosen and the matrices are equivalent then transformation matrices X and Y are also returned, such that $J = XHY$.

`HadamardMatrixToInteger(H): AlgMatElt -> RngIntElt`

Returns an integer that encodes the entries in the given Hadamard matrix in more compact form. The intended use is to save time when testing for equality against the same set of matrices many times.

`HadamardMatrixFromInteger(x, n): RngIntElt, RngIntElt -> AlgMatElt`

Returns a Hadamard matrix of degree n whose encoded form is the integer x. This function is the inverse of `HadamardMatrixToInteger()`.

Example: Hadamard Equivalence Testing (ex-709c48)

We demonstrate the use of some of these functions on certain degree 16 Hadamard matrices. For convenience, we create them from the more compact integer form.

```
> S := [
>   ␣
↪477589154830586526293008899241439041147987864578348429015179791084722816286 72,
>   ␣
↪525177435163503455147756353670354605777437302727377658527237928187550521708 05,
>   ␣
↪698092703726330756100475564287193740578698828040540591343460349699509316485 12,
>     720980122754871279613550713582055540325156009061483268413678201668044534 5792
> ];
> T := [ HadamardMatrixFromInteger(x, 16) : x in S ];
> &and [ IsHadamard(m) : m in T ];
true
```

Now we can test them for equivalence; we start by checking the 4-profiles.

```
> [ HadamardInvariant(m) : m in T ];
[
    [ 1680, 0, 140 ],
    [ 1680, 0, 140 ],
    [ 1344, 448, 28 ],
    [ 1344, 448, 28 ]
]
```

We see that the only possible equivalencies are between the first two and the last two, since equivalent matrices must have the same 4-profile.

```
> equiv,X,Y := IsHadamardEquivalent(T[1], T[2]);
> equiv;
true
> T[2] eq X*T[1]*Y;
true
> equiv,X,Y := IsHadamardEquivalent(T[3], T[4]);
> equiv;
false
```

So we have three inequivalent matrices. An alternative way to determine the inequivalent matrices would have been to use the canonical forms.

```
> #{ HadamardCanonicalForm(m) : m in T };
3
```

161.3 Associated 3-Designs

HadamardRowDesign(H, i): AlgMatElt, RngIntElt -> Dsgn

Given an $n \times n$ Hadamard matrix H (with $n \geq 4$) and an integer i with $1 \leq i \leq n$, returns the Hadamard 3–design corresponding to the ith row of H.

HadamardColumnDesign(H, i): AlgMatElt, RngIntElt -> Dsgn

Given an $n \times n$ Hadamard matrix H (with $n \geq 4$) and an integer i with $1 \leq i \leq n$, returns the Hadamard 3–design corresponding to the ith column of H.

Example: Hadamard Designs (ex-16b02c)

There is only one Hadamard equivalence class of 8×8 Hadamard matrices. We construct one matrix, and two of its designs.

```
> R := MatrixRing(Integers(), 8);
> H := R![1,  1,  1,  1,  1,  1,  1,  1,
>         1,  1,  1,  1, -1, -1, -1, -1,
>         1,  1, -1, -1,  1,  1, -1, -1,
>         1,  1, -1, -1, -1, -1,  1,  1,
>         1, -1,  1, -1,  1, -1, -1,  1,
>         1, -1,  1, -1, -1,  1,  1, -1,
>         1, -1, -1,  1, -1,  1, -1,  1,
>         1, -1, -1,  1,  1, -1,  1, -1];
> IsHadamard(H);
true
> DR := HadamardRowDesign(H, 3);
> DR: Maximal;
3-(8, 4, 1) Design with 14 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7, 8 @}
Blocks:
    {1, 2, 5, 6},
    {1, 2, 7, 8},
    {1, 2, 3, 4},
    {1, 4, 5, 7},
    {1, 4, 6, 8},
    {1, 3, 6, 7},
    {1, 3, 5, 8},
    {3, 4, 7, 8},
    {3, 4, 5, 6},
    {5, 6, 7, 8},
    {2, 3, 6, 8},
    {2, 3, 5, 7},
    {2, 4, 5, 8},
    {2, 4, 6, 7}
> HadamardColumnDesign(H, 8);
3-(8, 4, 1) Design with 14 blocks
```

161.4 Automorphism Group

HadamardAutomorphismGroup(H : parameters): AlgMatElt -> AlgMatElt

```
Al: MonStgElt                    Default: "Traces"
```

Given a Hadamard matrix H of degree n, returns the automorphism group of H as a permutation group of degree $4n$. The parameter `Al` may be set to one of `"Leon"`, `"nauty"`, or `"Traces"` (the default).

161.5 Databases

Magma contains two databases of Hadamard matrices; the first database includes all inequivalent matrices of degree at most 28, and examples of matrices of all degrees up to 256. The representatives used are the canonical forms (as output by `HadamardCanonicalForm`), and matrices of a given degree are ordered lexicographically (with 1 considered to be less than −1 for the purposes of this ordering).

A database of skew-symmetric Hadamard matrices also exists containing skew-symmetric Hadamard matrices of degrees 36, 44, and 52; in this case the matrices are not stored in canonical form, since the canonical forms are not skew-symmetric. With the exception of the creation routines, the intrinsics below apply to both databases.

`HadamardDatabase() -> DB`

Returns the database of Hadamard matrices.

`SkewHadamardDatabase() -> DB`

Returns the database of skew-symmetric Hadamard matrices.

`Matrix(D, n, k): DB, RngIntElt, RngIntElt -> AlgMatElt`

Returns the kth matrix of degree n in the database D.

`Matrices(D, n): DB, RngIntElt -> [ AlgMatElt ]`

Returns the sequence of all matrices of degree n that are stored in the database D.

`DegreeRange(D): DB -> RngIntElt, RngIntElt`

Returns the smallest and largest degrees of matrices in the database D.

`Degrees(D): DB -> [ RngIntElt ]`

Returns the sequence of degrees for which there is at least one matrix of that degree in the database D.

`NumberOfMatrices(D, n): DB, RngIntElt -> RngIntElt`

Return the number of matrices of degree n in the database D.

Example: Hadamard Matrix Database (ex-c34b5e)

```
> D := HadamardDatabase();
> Matrix(D, 16, 3);
[ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1]
[ 1  1  1  1  1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1]
[ 1  1  1  1 -1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1]
[ 1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1 -1]
[ 1  1 -1 -1 -1  1  1 -1  1 -1 -1  1 -1 -1  1  1]
[ 1  1 -1 -1  1 -1 -1  1 -1  1  1 -1 -1 -1  1  1]
[ 1  1 -1 -1 -1 -1  1  1 -1 -1  1  1  1  1 -1 -1]
[ 1 -1 -1  1 -1 -1  1  1  1  1 -1 -1  1 -1 -1  1]
[ 1 -1 -1  1  1  1 -1 -1 -1 -1  1  1  1 -1 -1  1]
[ 1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1]
[ 1 -1  1 -1  1 -1 -1  1  1 -1 -1  1 -1  1 -1  1]
[ 1 -1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1 -1  1]
[ 1 -1  1 -1 -1  1 -1  1 -1  1 -1  1  1 -1  1 -1]
[ 1 -1 -1  1  1 -1  1 -1 -1  1 -1  1 -1  1  1 -1]
[ 1 -1 -1  1 -1  1 -1  1  1 -1  1 -1 -1  1  1 -1]
[ 1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1  1  1  1  1]
> NumberOfMatrices(D, 20);
3
> Degrees(D);
[ 1, 2, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, 72,
76, 80, 84, 88, 92, 96, 100, 104, 108, 112, 116, 120, 124, 128, 132, 136, 140,
144, 148, 152, 156, 160, 164, 168, 172, 176, 180, 184, 188, 192, 196, 200, 204,
```

(continues on next page)

(continued from previous page)

```
208, 212, 216, 220, 224, 228, 232, 236, 240, 244, 248, 252, 256 ]
> [ NumberOfMatrices(D, n) : n in Degrees(D) ];
[ 1, 1, 1, 1, 1, 5, 3, 60, 487, 23, 218, 20, 500, 55, 562, 2, 3, 1, 2, 1, 2, 1,
3, 2, 1, 1, 1, 1, 2, 2, 1, 3, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
```

161.5.1 Updating the Databases

The databases are noticeably incomplete (for example, it is known that more than 60 000 Hadamard matrices exist for degree 32 whereas only 23 are in the database; also, only matrices of degrees 36, 44, or 52 are present in the skew database). The Magma group welcomes contributions of matrices not equivalent to ones already present and will add them to future versions of the databases.

The functions in this section can be used to create new versions of the databases. This allows users to include matrices not already present without having to wait for new official versions of the databases. These functions use a record whose format is not described, as this record should only be manipulated by these functions.

`HadamardDatabaseInformation(D : parameters): DB -> Rec`

```
Canonical: BoolElt                    Default: true
```

This takes an existing Hadamard database and extracts the information in it to an internal form which will be used by the other intrinsics. This internal form is returned. The parameter `Canonical` indicates whether the entries in the database are known to be the canonical forms or not. It defaults to `true`, which is correct for the standard database; it should be set to `false` if working with the skew database or a user-created database where the entries are not canonical.

Note that the value of the `Canonical` parameter also controls whether the database created from this data should store the canonical forms or the original ones. If you want to extract the matrices from a canonical database but store them in a non-canonical one, you should create the non-canonical database first — using either this intrinsic or `HadamardDatabaseInformationEmpty` — with `Canonical` set to `false` and then add the matrices from the database with `UpdateHadamardDatabase` and `Canonical` set to `true`.

`HadamardDatabaseInformationEmpty(: parameters) -> Rec`

```
Canonical: BoolElt                    Default: true
```

Returns the internal data that would correspond to an empty database. The parameter `Canonical` is used to indicate whether the entries in the database should be written out in canonical form or not.

This allows the creation of a new database, or a slice of an existing one, without including the entirety of a previous one.

`UpdateHadamardDatabase(~R, S : parameters): Rec, SeqEnum`

```
Canonical: BoolElt                    Default: false
```

This augments the record R containing the database information with the matrices in the sequence S. The matrices are only added if they are inequivalent with matrices already there. Note that this requires finding the canonical forms, which can be expensive. If the matrices in S are known to be in canonical form already then the parameter `Canonical` should be set to `true`.

If the matrices of matching degree already in R are not known to be canonical then their canonical forms must also be computed. Again, this can take significant time. This is particularly troublesome for the skew database, where the canonical forms are not stored. For one way to deal with this, see the description of `WriteRawHadamardData` below.

`WriteHadamardDatabase(S, ~R): MonStgElt, Rec`

This creates the database files *name*`.dat` and *name*`.ind` from the database data R, where *name* is taken from the string S. Since canonical forms may need to be computed during this process, the data is passed with a variable reference so that this computation is not lost. (For instance, if one writes out the database, then adds some more matrices and writes it out again.)

`WriteRawHadamardData(S, R): MonStgElt, Rec`

This saves the data in R to the file whose name is given by the string S. When loaded, this file will define a single variable `data` which will behave identically to R. This is desirable for the non-canonical databases since the canonical forms will not have to be recomputed.

This routine destroys the original contents, if any, of the file.

`SetVerbose("HadamardDB", v): MonStgElt, RngIntElt`

This procedure changes the verbose printing level for the Hadamard database update routines. The verbose value v should be an integer in the range 0 to 3. This can provide reassuring indications of progress when a long update process is underway.

Example: Updating the Hadamard Database (ex-fc270d)

Here are some examples of how one might use these routines. We assume that the file `matrixfile` contains code that will define a sequence S of Hadamard matrices when loaded.

```
> D := HadamardDatabase();
> #D;
5391
> data := HadamardDatabaseInformation(D);
> SetVerbose("HadamardDB", 1);
> load "matrixfile";
> UpdateHadamardDatabase(~data, S);
3 new matrices added
> WriteHadamardDatabase("~/data/hadamard", ~data);
Sorting matrices by canonical forms
Sorting by degrees
Computing size information
Writing data file
Data file written (2007 matrices total)

Writing index file
Index file written
```

This has created the files `hadamard.dat` and `hadamard.ind` (in the directory `~/data`) containing the updated data for the Hadamard database. The file stems should be `hadamard` for the standard database, or `hadamard_skew` for the database of skew-symmetric matrices; in this case we wanted the former. In order to use these files instead of the standard database, the library root must be changed. Database files are looked for in the `data` subdirectory of the library root, so in this example the library root should be changed to `~` in order to get the newly created database.

```
> SetLibraryRoot("~");
> D := HadamardDatabase();
> #D;
2007
```

Of course, the library root is used by other databases, so it would be a good idea to set it back to its original value after creating the modified database. A better approach would be to encapsulate this process in a function (which could be put into the user startup file).

```
> function MyHadamardDatabase()
>     oldlibroot := GetLibraryRoot();
```

(continues on next page)

(continued from previous page)

```
>     SetLibraryRoot("~");
>     D := HadamardDatabase();
>     SetLibraryRoot(oldlibroot);
>     return D;
> end function;
>
> D := MyHadamardDatabase();
> #D;
2007
```

If working with the database of skew-symmetric Hadamard matrices, it will probably be desirable to use the raw data instead, since the computation of the canonical forms takes a lot of time (about 23 minutes on a 750MHz machine). This does have to be done, but by using the raw forms it will only have to be done once, rather than each time.

```
> D := SkewHadamardDatabase();
> #D;
638
> data := HadamardDatabaseInformationEmpty(: Canonical := false);
> for n in Degrees(D) do
>     UpdateHadamardDatabase(~data, Matrices(D, n));
> end for;
> WriteRawHadamardData("skewraw.m", data);
```

Now that the raw data has been created, it can be used in other sessions to update the database (or the same session, of course).

```
> load "skewraw.m";
> load "matrixfile";
> UpdateHadamardDatabase(~data, S);
> WriteHadamardDatabase("~/data/hadamard_skew", ~data);
> WriteRawHadamardData("skewraw.m", data);
```

Note that the updated raw form is saved as well as the database. Now the new data can be accessed in the same way as the previous database was.

```
> SetLibraryRoot("~");
> D := SkewHadamardDatabase();
> #D;
641
```

As before, for maximum convenience access to the new database should also be put into a function in the startup file.

Chapter 162

Graphs

162.1 Introduction

A *simple graph* is a graph in which each edge joins two distinct vertices and two distinct vertices are joined by at most one edge. A *simple digraph*, whose edges are directed, is defined in an analogous manner. Thus, loops and multiple edges are not permitted in simple graphs and simple digraphs. A graph (digraph) with loops and/or multiple edges joining a fixed pair of vertices is called a *multigraph* (resp. *multidigraph*). A multidigraph whose edges are assigned a capacity is more commonly called a *network*.

In this chapter the term "graph" is used when referring to a simple undirected graph, while the term "digraph" is used when referring to a simple directed graph. Sometimes the term "graph" is used as the generic term for the incidence structure on vertices and edges. Such uses should be clear from the context in which they occur.

There are five Magma graph objects: the undirected simple graph of type `GrphUnd`, the directed simple graph of type `GrphDir`, the undirected multigraph of type `GrphMultUnd`, the directed multigraph of type `GrphMultDir`, and the network of type `GrphNet`. The simple graphs are all of type `Grph`, while the multigraphs (including the network) are of type `GrphMult`. There is a caveat here with respect to simple digraphs and loops: for historical reasons, Magma allows loops in digraphs.

Simple graphs and digraphs are covered in this chapter, while multigraphs, multidigraphs are covered in Chapter *Multigraphs* and networks in Chapter *Networks*. All types of graphs may have vertex labellings and/or edge labellings. In addition, assigning weights and/or capacities to graph edges is also possible, thus allowing to run shortest-paths and flow-based algorithms over the graphs.

Importantly, from the present version (V2.11) onwards, all the standard graph construction functions (see Subsections *Subgraphs and Quotient Graphs* and *Incremental Construction of Graphs*) respect the graph's support set and vertex and edge decorations. That is, the resulting graph will have a support set and vertex and edge decorations compatible with the original graph and the operation performed on that graph.

Magma employs two distinct data structures for representing graphs. A graph may be represented in the form of an adjacency matrix (the dense representation) or in the form of an adjacency list (the sparse representation). The latter is better suited for sparse graphs and for algorithms which have been designed with the adjacency list representation in mind. An advantage of the sparse representation is the possibility of creating much larger (sparse) graphs than would be possible using the dense representation, since memory requirements for the adjacency list representation is linear in the number of edges, while memory requirements for the adjacency matrix representation is quadratic in the number of vertices (order) of the graph. This is covered in detail in Section *Bounds on the Graph Order*.

Further, multigraphs and multidigraphs (and networks) may only be represented by an adjacency list since they may contain multiple edges. Users have control over the choice of representation when creating simple graphs or digraphs. If no indication is given, simple graphs and digraphs are *always* created with the dense representation. At the time of the present release (V2.11), a significant part of Magma functions

are able to work directly with either of the representations. However, wherever necessary, a graph will be converted internally to whichever representation is required by a given function. We emphasize that this process is completely transparent to users and *requires no intervention on their part.*

For a concise and comprehensive overview on graph theory and algorithms, the reader is referred to [Eve79]; in [TCR90] they'll find a clear exposure of some graph algorithms, and [AMO93] is the recommended reference for flow problems in graphs.

162.2 Construction of Graphs and Digraphs

Any enumerated or indexed set S may be given as the vertex-set of a graph. The graph constructor will take a copy V of S, convert V into an indexed set if necessary, and flag its type as `GrphVertSet`. A graph may be specifically created as a sparse graph. If no indication is given then the graph is *always* created with the dense representation, that is, as an adjacency matrix.

162.2.1 Bounds on the Graph Order

Memory allocation for any elementary object (that is, accessed via a single pointer) cannot exceed 2^{32} bytes. For this reason there is a bound on the graph order. This bound is dependent upon the (internal) graph representation chosen at creation: the dense representation (as a packed adjacency matrix), or the sparse representation (as an adjacency list). The former is quadratic in the number of vertices of the graph, the latter is linear in the number of the edges. Thus a large graph with a low edge density will require less memory space than the same graph with a dense representation.

The bounds on the graph order n are as follows:

- for the dense representation, $n \leq 65535$,

- for the sparse representation, $n \leq 134217722$.

These bounds are maximal, that is, they assume — for the sparse representation — that the number of edges is zero. To help users determine the likely size of the graph they want to construct, we provide the following function.

`GraphSizeInBytes(n, m : parameters): RngIntElt, RngIntElt -> RngIntElt`

```
IsDigraph: Bool                    Default: false
SparseRep: Bool                    Default: false
```

Computes the memory requirement in bytes of a graph of order n and size m. By default it is assumed that the graph is undirected and has a dense representation.

Example: Size Limit of a Dense Graph `(ex-1d9be6)`

One may verify that the dense representation cannot be used for a value of n (graph order) larger than 65535.

```
> n := 65536;
> m := 0;
> assert GraphSizeInBytes(n, m) gt 2^32;
```

It is possible to construct such a graph with a sparse representation:

```
> GraphSizeInBytes(n, m : SparseRep := true);
2097580
```

However, assuming that the graph of order $n = 65535$ has a size of $m = 33538101$ we see that such a graph cannot be constructed as its total memory requirement is again larger than 2^{32}:

```
> m := 33538101;
> assert GraphSizeInBytes(n, m : SparseRep := true) gt 2^32;
```

162.2.2 Construction of a General Graph

`Graph< n | edges : parameters>: RngIntElt, List -> GrphUnd, GrphVertSet, GrphEdgeSet`
`Graph< S | edges : parameters >: SetEnum, List -> GrphUnd, GrphVertSet, GrphEdgeSet`
`Graph< S | edges : parameters>: SetIndx, List -> GrphUnd, GrphVertSet, GrphEdgeSet`

`SparseRep: Bool` `Default: false`

Construct the graph G with vertex-set $V = \{@v_1, v_2, \ldots, v_n@\}$ (where $v_i = i$ for each i if the first form of the constructor is used, or the ith element of the enumerated or indexed set S otherwise), and edge-set $E = \{e_1, e_2, \ldots, e_q\}$. This function returns three values: The graph G, the vertex-set V of G; and the edge-set E of G. If SparseRep is `true` then the resulting graph will have a sparse representation.

The elements of E are specified by the list `edges`, where the items of `edges` may be objects of the following types:

(a) A pair $\{v_i, v_j\}$ of vertices in V. The edge from v_i to v_j will be added to the edge-set for G.

(b) A tuple of the form $\langle v_i, N_i\rangle$ where N_i will be interpreted as a set of neighbours for the vertex v_i. The elements of the sets N_i must be elements of V. If $N_i = \{u_1, u_2, \ldots, u_r\}$, the edges $\{v_i, u_1\}, \ldots, \{v_i, u_r\}$ will be added to G.

(c) A sequence $[N_1, N_2, \ldots, N_n]$ of n sets, where N_i will be interpreted as a set of neighbours for the vertex v_i. The edges $\{v_i, u_i\}$, $1 \leq i \leq n$, $u_i \in N_i$, are added to G.

In addition to these three basic ways of specifying the *edges* list, the items in *edges* may also be:

(d) An edge e of a graph or digraph or multigraph or multidigraph or network of order n. If e is an edge from u to v in a directed graph H, then the undirected edge $\{u, v\}$ will be added to G.

(e) An edge-set E of a graph or digraph or multigraph or multidigraph or network of order n. Every edge e in E will be added to G.

(f) A graph or a digraph or a multigraph or a multidigraph or a network H of order n. Every edge e in H's edge-set is added to G.

(g) A $n \times n$ symmetric (0, 1)-matrix A. The matrix A will be interpreted as the adjacency matrix for a graph H on n vertices and the edges of H will be added will be added to G.

(h) A set of

(i) Pairs of the form $\{v_i, v_j\}$ of vertices in V.

(ii) Tuples of the form $\langle v_i, N_i\rangle$ where N_i will be interpreted as a set of neighbours for the vertex v_i.

(iii) Edges of a graph or digraph or multigraph or multidigraph or network of order n.

(iv) Graphs or digraphs or multigraphs or multidigraphs or networks of order n.

(j) A sequence of

(i) Tuples of the form $\langle v_i, N_i\rangle$ where N_i will be interpreted as a set of neighbours for the vertex v_i.

`IncidenceGraph(A): ModMatRngElt -> GrphUnd`

Let A be a $n \times m$ $(0, 1)$ matrix having exactly two 1s in each column. Then a graph G of order n and size m having A as its incidence matrix will be constructed. The rows of A will correspond to the vertices of G while the columns will correspond to the edges.

Example: Constructing the Petersen Graph (ex-f07f4d)

The Petersen graph will be constructed in various different ways to illustrate the use of the constructor. Firstly, it will be constructed by listing the edges:

```
> P := Graph< 10 | { 1, 2 }, { 1, 5 }, { 1, 6 }, { 2, 3 }, { 2, 7 },
>            { 3, 4 }, { 3, 8 }, { 4, 5 }, { 4, 9 }, { 5, 10 },
>            { 6, 8 }, { 6, 9 }, { 7, 9 }, { 7, 10 }, { 8, 10 } >;
```

We next construct the Petersen graph by listing the neighbours of each vertex:

```
> P := Graph< 10 | [ { 2, 5, 6 }, { 1, 3, 7 }, { 2, 4, 8 }, { 3, 5, 9 },
>               { 1, 4, 10 }, { 1, 8, 9 }, { 2, 9, 10 }, { 3, 6, 10 },
>               { 4, 6, 7 }, { 5, 7, 8 } ] >;
```

We repeat the above construction but now the graph is created as a sparse graph:

```
> PS := Graph< 10 | [ { 2, 5, 6 }, { 1, 3, 7 }, { 2, 4, 8 }, { 3, 5, 9 },
>               { 1, 4, 10 }, { 1, 8, 9 }, { 2, 9, 10 }, { 3, 6, 10 },
>               { 4, 6, 7 }, { 5, 7, 8 } ] : SparseRep := true >;
> assert PS eq P;
```

Here is yet another way of constructing the sparse graph, from the dense graph:

```
> PS := Graph< 10 | P : SparseRep := true >;
> assert PS eq P;
```

We finally construct the graph in terms of an adjacency matrix:

```
> M := MatrixRing( Integers(), 10 );
> P := Graph< 10 | M![ 0,1,0,0,1,1,0,0,0,0,
>                   1,0,1,0,0,0,1,0,0,0,
>                   0,1,0,1,0,0,0,1,0,0,
>                   0,0,1,0,1,0,0,0,1,0,
>                   1,0,0,1,0,0,0,0,0,1,
>                   1,0,0,0,0,0,0,1,1,0,
>                   0,1,0,0,0,0,0,0,1,1,
>                   0,0,1,0,0,1,0,0,0,1,
>                   0,0,0,1,0,1,1,0,0,0,
>                   0,0,0,0,1,0,1,1,0,0] >;
> P;

       Graph
   Vertex  Neighbours

      1     2 5 6 ;
      2     1 3 7 ;
      3     2 4 8 ;
      4     3 5 9 ;
      5     1 4 10 ;
      6     1 8 9 ;
      7     2 9 10 ;
      8     3 6 10 ;
      9     4 6 7 ;
     10     5 7 8 ;
```

Example: Tutte's 8-Cage (ex-d2036c)

A more sophisticated example is the construction of Tutte's 8-cage using the technique described in P. Lorimer [Lor89]. The graph is constructed so that it has $G = P\Gamma L(2, 9)$ in its representation

of degree 30 as its automorphism group. The vertices of the graph correspond to the points on which G acts. The neighbours of vertex 1 are the points lying in the unique orbit N_1 of length 3 of the stabilizer of 1. The edges for vertex i are precisely the points N_1^g where g is an element of G such that $1^g = i$.

```
> G := PermutationGroup< 30 |
>     (1, 2)(3, 4)(5, 7)(6, 8)(9, 13)(10, 12)(11, 15)(14, 19) (16, 23)
>         (17, 22)(18, 21)(20, 27)(24, 29)(25, 28)(26, 30),
>     (1, 24, 28, 8)(2, 9, 17, 22)(3, 29, 19, 15)(4, 5, 21, 25)
>         (6, 18, 7, 16)(10, 13, 30, 11)(12, 14)(20, 23)(26, 27) >;
> N1 := rep{ o : o in Orbits(Stabilizer(G, 1)) | #o eq 3 };
> tutte := Graph< 30 | <1, N1>^G >;
```

162.2.3 Construction of a General Digraph

`Digraph< n | edges : parameters>: RngIntElt, List -> GrphDir`

`Digraph< S | edges : parameters>: SetEnum, List -> GrphDir`

`Digraph< S | edges : parameters>: SetIndx, List -> GrphDir`

```
SparseRep: Bool                    Default: false
```

Construct the digraph G with vertex-set $V = \{@v_1, v_2, \ldots, v_n@\}$ (where $v_i = i$ for each i if the first form of the constructor is used, or the ith element of the enumerated or indexed set S otherwise), and edge-set $E = \{e_1, e_2, \ldots, e_q\}$. This function returns three values: The graph G, the vertex-set V of G; and the edge-set E of G. If SparseRep is `true` then the resulting graph will have a sparse representation.

The elements of E are specified by the list `edges`, where the items of `edges` may be objects of the following types:

(a) A pair $[v_i, v_j]$ of vertices in V. The directed edge from v_i to v_j will be added to the edge-set for G.

(b) A tuple of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i. The elements of the sets N_i must be elements of V. If $N_i = \{u_1, u_2, \ldots, u_r\}$, the directed edges $[v_i, u_1], \ldots, [v_i, u_r]$ will be added to G.

(c) A sequence $[N_1, N_2, \ldots, N_n]$ of n sets, where N_i will be interpreted as a set of neighbours for the vertex v_i. The directed edges $[v_i, u_i]$, $1 \leq i \leq n$, $u_i \in N_i$, are added to G.

In addition to these three basic ways of specifying the *edges* list, the items in *edges* may also be:

(d) An edge e of a graph or digraph or multigraph or multidigraph or network of order n. If e is an edge $\{u, v\}$ in an undirected graph then both directed edges $[u, v]$ and $[v, u]$ are added to G.

(e) An edge-set E of a graph or digraph or multigraph or multidigraph or network of order n. Every edge e in E will be added to G according to the rule set out for a single edge.

(f) A graph or a digraph or a multigraph or a multidigraph or a network H of order n. Every edge e in H's edge-set is added to G according to the rule set out for a single edge.

(g) A $n \times n$ (0, 1)-matrix A. The matrix A will be interpreted as the adjacency matrix for a digraph H on n vertices and the edges of H will be added will be added to G.

(h) A set of

(i) Pairs of the form $[v_i, v_j]$ of vertices in V.

(ii) Tuples of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i.

(iii) Edges of a graph or digraph or multigraph or multidigraph or network of order n.

(iv) Graphs or digraphs or multigraphs or multidigraphs or networks of order n.

(j) A sequence of

(i) Tuples of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i.

`IncidenceDigraph(A): ModMatRngElt -> GrphDir`

Let A be a $n \times m$ matrix such that each column contains at most one entry equal to $+1$ and at most one entry equal to -1 (all others being zero). Then a digraph G of order n and size m having A as its incidence matrix will be constructed. The rows of A will correspond to the vertices of G while the columns will correspond to the edges. If column k of A contains $+1$ in row i and -1 in row j, the directed edge $v_i v_j$ will be included in G. If only row i has a non-zero entry (either 1 or -1), then the loop $v_i v_i$ will be included in G.

Example: Constructing a Digraph (ex-fe5c22)

The digraph D with vertices $\{v_1, v_2, v_3, v_4, v_5\}$ and edges $(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_3, v_2)$, and (v_4, v_3) will be constructed, firstly by listing its edges and secondly by giving its adjacency matrix.

```
> D1 := Digraph< 5 | [1, 2], [1, 3], [1, 4],
>                   [3, 2], [4, 3] >;
> D1;
Digraph
Vertex  Neighbours

1       2 3 4 ;
2       ;
3       2 ;
4       3 ;
5       ;
```

The same digraph with a sparse representation:

```
> D1 := Digraph< 5 | [1, 2], [1, 3], [1, 4],
>                   [3, 2], [4, 3] : SparseRep := true>;
```

We next construct the digraph by giving its adjacency matrix:

```
> M := MatrixRing(Integers(), 5);
> D2 := Digraph< 5 |
>           M![ 0,1,1,1,0,
>               0,0,0,0,0,
>               0,1,0,0,0,
>               0,0,1,0,0,
>               0,0,0,0,0 ] >;
> IsIsomorphic(D1, D2);
true
```

162.2.4 Operations on the Support

Let S be the set over which the graph G is defined. The set S is referred to as the *support* of G.

`Support(G): Grph -> SetIndx`

`Support(V): GrphVertSet -> SetIndx`

The indexed set used in the construction of G (or the graph for which V is the vertex-set), or the standard set $\{@1, \ldots, n@\}$ if it was not given.

ChangeSupport(G, S): Grph, SetIndx -> Grph, GrphVertSet, GrphEdgeSet

If G is a graph having n vertices and S is an indexed set of cardinality n, return a new graph H equal to G but whose support is S. That is, H is structurally equal to G and its vertex and edge decorations are the *same* as those for G.

ChangeSupport(~G, S): Grph, SetIndx

The procedural version of the above function.

StandardGraph(G): Grph -> Grph

The graph H equal to G but defined on the standard support. That is, H is structurally equal to G and its vertex and edge decorations are the same as those for G.

Example: Standard Graph of an Odd Graph (ex-ec4c86)

The Odd Graph, O_n, has as vertices all n-subsets of a $2n-1$ set with two vertices adjacent if and only if they are disjoint. We construct O_3 and then form its standard graph.

```
> V := Subsets({1..5}, 2);
> O3 := Graph< V | { {u,v} : u,v in V | IsDisjoint(u, v) } >;
> O3;
Graph
Vertex  Neighbours

{ 1, 5 }        { 2, 4 } { 2, 3 } { 3, 4 } ;
{ 2, 5 }        { 1, 3 } { 1, 4 } { 3, 4 } ;
{ 1, 3 }        { 2, 5 } { 2, 4 } { 4, 5 } ;
{ 1, 4 }        { 2, 5 } { 3, 5 } { 2, 3 } ;
{ 2, 4 }        { 1, 5 } { 1, 3 } { 3, 5 } ;
{ 3, 5 }        { 1, 4 } { 2, 4 } { 1, 2 } ;
{ 2, 3 }        { 1, 5 } { 1, 4 } { 4, 5 } ;
{ 1, 2 }        { 3, 5 } { 3, 4 } { 4, 5 } ;
{ 3, 4 }        { 1, 5 } { 2, 5 } { 1, 2 } ;
{ 4, 5 }        { 1, 3 } { 2, 3 } { 1, 2 } ;

> Support(O3);
{@
    { 1, 5 },
    { 2, 5 },
    { 1, 3 },
    { 1, 4 },
    { 2, 4 },
    { 3, 5 },
    { 2, 3 },
    { 1, 2 },
    { 3, 4 },
    { 4, 5 }
@}
> SO3 := StandardGraph(O3);
> SO3;
Graph
Vertex  Neighbours

1       5 7 9 ;
2       3 4 9 ;
3       2 5 10 ;
4       2 6 7 ;
5       1 3 6 ;
6       4 5 8 ;
7       1 4 10 ;
```

(continues on next page)

(continued from previous page)

```
8       6 9 10 ;
9       1 2 8 ;
10      3 7 8 ;
> Support(SO3);
{@ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 @}
```

162.2.5 Construction of a Standard Graph

Some of the construction functions listed below can also be used to create a graph with a sparse representation. Are concerned only those functions creating relatively sparse graphs.

BipartiteGraph(m, n): RngIntElt, RngIntElt -> GrphUnd

The complete bipartite graph, $K_{m,n}$, with partite sets of cardinality m and n.

CompleteGraph(n): RngIntElt -> GrphUnd

The complete graph K_n on n vertices.

KCubeGraph(n : parameters): RngIntElt -> GrphUnd

```
SparseRep: Bool                    Default: false
```

The graph of the n-dimensional cube, Q_n. The graph can be created as a sparse graph if so requested.

MultipartiteGraph(Q): [RngIntElt] -> GrphUnd

Given a sequence Q of positive integers $m_1, \ldots, m_r$, construct the complete multipartite graph $K_{m_1,\ldots,m_r}$.

EmptyGraph(n : parameters): RngIntElt -> GrphUnd

```
SparseRep: Bool                    Default: false
```

The graph on n vertices with no edges. It can be created as a sparse graph if so requested.

NullGraph(: parameters) -> GrphUnd

```
SparseRep: Bool                    Default: false
```

The graph with no vertices and no edges. It can be created as a sparse graph if so requested.

PathGraph(n : parameters): RngIntElt -> GrphUnd

```
SparseRep: Bool                    Default: false
```

The path graph on n vertices, i.e. the graph on vertices $v_1, \ldots, v_n$, with v_i and v_j $(1 \leq i < j \leq n)$ adjacent if and only if $j = i + 1$. The graph can be created as a sparse graph if so requested.

PolygonGraph(n : parameters): RngIntElt -> GrphUnd

```
SparseRep: Bool                    Default: false
```

Given an integer $n \geq 3$, define the polygon graph on n vertices, i.e. the graph on vertices $v_1, \ldots, v_n$, with v_i and v_j $(1 \leq i < j \leq n)$ adjacent if and only if $j = i + 1$, or $i = 1$ and $j = n$. The graph can be created as a sparse graph if so requested.

RandomGraph(n, r : parameters): RngIntElt, FldReElt -> GrphUnd

```
SparseRep: Bool                    Default: false
```

A random graph on n vertices such that the probability that an arbitrary pair of vertices is adjacent is given by the real number r, where r lies in the interval $[0, 1]$. The graph can be created as a sparse graph if so requested.

`RandomTree(n : parameters): RngIntElt -> GrphUnd`

```
SparseRep: Bool                    Default: false
```

A random tree on n vertices. It can be created as a sparse graph if so requested.

162.2.6 Construction of a Standard Digraph

As for standard graphs (Subsection *Construction of a Standard Graph*) some of the construction functions listed below can be used to create digraphs with a sparse representation.

`CompleteDigraph(n): RngIntElt -> GrphDir`

The complete symmetric digraph on n vertices.

`EmptyDigraph(n : parameters): RngIntElt -> GrphDir`

```
SparseRep: Bool                    Default: false
```

The null digraph on n vertices. It can be created as sparse if so requested.

`RandomDigraph(n, r : parameters): RngIntElt, FldReElt -> GrphDir`

```
SparseRep: Bool                    Default: false
```

A random digraph on n vertices such that the probability that there exists an edge directed from vertex u to vertex v, where u and v arbitrary vertices, is given by the real number r, where r lies in the interval $[0, 1]$. The digraph can be created as sparse if so requested.

Example: Complete and Random Digraphs (ex-d57426)

The complete symmetric digraph on 5 vertices is created as follows:

```
> kd5 := CompleteDigraph(5);
> kd5;
Digraph
Vertex  Neighbours

1       2 3 4 5 ;
2       1 3 4 5 ;
3       1 2 4 5 ;
4       1 2 3 5 ;
5       1 2 3 4 ;
```

We next construct a random digraph on 5 vertices such that the probability that there is an edge directed from an arbitrary vertex to any other arbitrary vertex is 0.75.

```
> rd5 := RandomDigraph(5, 0.75);
> rd5;
Digraph
Vertex  Neighbours

1       2 5 ;
2       2 3 4 5 ;
3       2 3 4 5 ;
4       1 2 3 5 ;
5       2 3 4 ;
```

162.3 Graphs with a Sparse Representation

As mentioned in the Introduction *Introduction* of this chapter it is possible to construct graphs having a sparse representation. That is, graphs which are represented by means of an adjacency list. This has an obvious advantage for graphs with a low edge density, in that it allows to construct much larger graphs than if they were represented by means of an adjacency matrix (the dense representation). See Section *Bounds on the Graph Order* for more details on this issue.

Another advantage of the sparse representation is for graph algorithms which are linear in the number of edges (the planarity tester and the triconnectivity tester), and more generally, for those algorithms based on the adjacency list representation (the flow-based algorithms and the shortest-paths algorithms). Further, the sparse representation is required when creating multigraphs (see Chapter *Multigraphs*) since they may have multiple edges.

A significant number, but not all, of the existing Magma functions have been written for both representations. Should a function designed for a matrix representation of the graph be applied to a graph with a sparse representation, then the graph is automatically converted *without any user intervention*. This is also true when the reverse conversion is required.

Without being exhaustive, we will list here the functions which can deal with both graph representations without having to perform an (internal) conversion.

- Nearly all construction functions (*Construction of Graphs and Digraphs*),
- All functions in *The Vertex–Set and Edge–Set of a Graph*, *Subgraphs and Quotient Graphs*, *Incremental Construction of Graphs*,
- Some functions in *Constructing Complements, Line Graphs; Contraction, Switching* and *Unions and Products of Graphs*,
- All functions in *Converting between Graphs and Digraphs*,
- The basic invariants and predicates in *Elementary Invariants of a Graph* and *Elementary Graph Predicates*,
- Almost all functions in *Adjacency and Degree*,
- All functions in *Connectedness in a Graph* and all but the last in *Connectedness in a Digraph*,
- The functions in *Distances, Paths, and Circuits in a Possibly Weighted Graph*,
- All functions in *Spanning Trees of a Graph or Digraph*.

The functions in *Labelled, Capacitated, and Weighted Graphs*, *Graph Triconnectivity*, *Maximum Matching in Bipartite Graphs*, *General Vertex and Edge Connectivity in Graphs and Digraphs*, and *Planar Graphs*, and all the functions listed in Chapter *Multigraphs*, deal with a graph as an adjacency list.

The functions in *Construction from Groups, Codes, and Designs*, *Distances, Paths, and Circuits in a Non-Weighted Graph*, *Matrices and Vector Spaces Associated with a Graph or Digraph*, *Directed Trees*, *Colourings*, *Cliques, Independent Sets*, *Automorphism Group of a Graph or Digraph*, and *Symmetry and Regularity Properties of Graphs* deal with a graph as an adjacency matrix.

We emphasize again that should a conversion of the graph's representation take place due to internal specifications, the conversion takes place *automatically*, that is, *without user intervention*.

The only problem that may occur is when such a conversion concerns a (very) large graph and thus may possibly result in a lack of memory to create the required representation. This is where users might require control over which internal representation is used so as to minimise the likelihood of both representations coexisting. This is achieved by tuning the creation functions (as described in Section *Construction of Graphs and Digraphs*), and the following functions which determine the nature of a graph's representation.

When conversion of the graph's representation into the alternative one occurs the original representation is not deleted.

```
HasSparseRep(G): Grph -> BoolElt
```

```
HasDenseRep(G): Grph -> BoolElt
```

HasSparseRepOnly(G): Grph -> BoolElt

HasDenseRepOnly(G): Grph -> BoolElt

HasDenseAndSparseRep(G): Grph -> BoolElt

This set of functions determine the nature of a graph's representation.

Example: Dense and Sparse Representations (ex-b3b8b8)

We give four examples, each illustrating one of the four possible cases that may arise. First, when the graph's dense representation remains unchanged:

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} }>;
> HasDenseRepOnly(G);
true
> A := AutomorphismGroup(G);
> HasDenseRepOnly(G);
true
```

The same example using a sparse graph representation:

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} } : SparseRep := true>;
> HasSparseRepOnly(G);
true
> A := AutomorphismGroup(G);
> HasDenseAndSparseRep(G);
true
```

Next the case when the graph's sparse representation remains unchanged:

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} } : SparseRep := true>;
> HasSparseRepOnly(G);
true
> IsPlanar(G);
true
> HasSparseRepOnly(G);
true
```

Finally the same example using a dense graph representation:

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} }>;
> HasDenseRepOnly(G);
true
> IsPlanar(G);
true
> HasDenseAndSparseRep(G);
true
```

162.4 The Vertex–Set and Edge–Set of a Graph

162.4.1 Introduction

Let G be a graph on n vertices and m edges whose vertex-set is $V = \{v_1, \ldots, v_m\}$ and edge-set is $E = \{e_1, \ldots, e_m\}$. A graph created by Magma consists of three objects: the *vertex-set* V, the *edge-set* E, and the graph G itself. The vertex-set and edge-set of a graph are *enriched* sets and consequently constitute types. The vertex-set and edge-set are returned as the second and third arguments, respectively, by all functions which create graphs. Alternatively, a pair of functions are provided to extract the vertex-set and edge-set of a graph G. The main purpose of having vertex-sets and edge-sets as types is to provide a convenient mechanism for referring to vertices and edges of a graph. Here, the functions applicable to vertex-sets and edge-sets are described.

162.4.2 Creating Edges and Vertices

`EdgeSet(G): Grph -> GrphEdgeSet`

Given a graph G, return the edge-set of G.

`Edges(G): Grph -> {@ GrphEdge @}`

A set E whose elements are the edges of the graph G. Note that this creates an indexed set and not the edge-set of G, in contrast to the function `EdgeSet`.

`VertexSet(G): Grph -> GrphVertSet`

Given a graph G, return the vertex-set of G.

`Vertices(G): Grph -> { GrphVert}`

A set V whose elements are the vertices of the graph G. In contrast to the function `VertexSet`, this function returns the collection of vertices of G in the form of an indexed set.

`V ! v: GrphVertSet, . -> GrphVert`

Given the vertex-set V of the graph G, coerce v to a vertex of G. If v lies in the support of G, return the corresponding vertex. If v is an integer i with $1 \leq i \leq \#V$, return the vertex with index i (equivalent to `V . i`).

`V . i: GrphVertSet, RngIntElt -> GrphVert`

Given a vertex-set V and an integer i such that $1 \leq i \leq \#V$, create the vertex v_i of V.

`Index(v): GrphVert -> RngIntElt`

Given a vertex v of some graph G, return the index of v in the (indexed) vertex-set of G.

`E ! { u, v}: GrphEdgeSet, { . } -> GrphEdge`

Given the edge-set E of an *undirected* graph G, coerce the unordered pair $\{u, v\}$ to the edge of G joining the vertices $V!u$ and $V!v$, where V is the vertex-set of G. It is an error to use this form on a directed graph; use `E ! [u, v]` for directed graphs.

`E ! [u, v]: GrphEdgeSet, [ . ] -> GrphEdge`

Given the edge-set E of a *directed* graph G (not a network), coerce the ordered pair $[u, v]$ to the directed edge of G from vertex $V!u$ to vertex $V!v$, where V is the vertex-set of G. It is an error to use this form on an undirected graph; use `E ! {u, v}` for undirected graphs.

`E . i: GrphEdgeSet, RngIntElt -> GrphEdge`

Given an edge-set E and an integer i such that $1 \leq i \leq \#E$, create the i-th edge of E.

Example: Vertices and Edges of an Odd Graph (ex-e2c41e)

The construction of vertices and edges is illustrated using the Odd Graph, O_3. and then form its standard graph.

```
> S := Subsets({1..5}, 2);
> O3, V, E := Graph< S | { {u,v} : u,v in S | IsDisjoint(u, v) } >;
> VertexSet(O3);
Vertex-set of O3
> Vertices(O3);
{@ { 1, 5 }, { 2, 5 }, { 1, 3 }, { 1, 4 }, { 2, 4 }, { 3, 5 },
{ 2, 3 }, { 1, 2 }, { 3, 4 }, { 4, 5 } @}
> EdgeSet(O3);
Edge-set of O3
> Edges(O3);
{@ {{ 1, 5 }, { 2, 4 }}, {{ 1, 5 }, { 2, 3 }}, {{ 1, 5 }, { 3, 4 }},
{{ 2, 5 }, { 1, 3 }}, {{ 2, 5 }, { 1, 4 }}, {{ 2, 5 }, { 3, 4 }},
{{ 1, 3 }, { 2, 4 }}, {{ 1, 3 }, { 4, 5 }}, {{ 1, 4 }, { 3, 5 }},
{{ 1, 4 }, { 2, 3 }}, {{ 2, 4 }, { 3, 5 }}, {{ 3, 5 }, { 1, 2 }},
{{ 2, 3 }, { 4, 5 }}, {{ 1, 2 }, { 3, 4 }}, {{ 1, 2 }, { 4, 5 }} @}
> u := V!{1, 2};
```

(continues on next page)

(continued from previous page)

```
> u, Type(u);
{1, 2} GrphVert
> Index(u);
8
> V ! 8;
{1, 2} GrphVert
> x := E!{ {1,2}, {3,4}};
> x, Type(x);
{{ 1, 2 }, { 3, 4 }} GrphEdge
```

Example: Coercing Vertices and Edges of a Digraph (ex-12cef7)

Directed edge coercion uses sequence notation.

```
> D, VD, ED := Digraph< 3 | {[1,2],[2,3],[3,1]} >;
> v := VD ! 2;
> v, Type(v);
2 GrphVert
> e := ED ! [1, 2];
> e, Type(e);
[1, 2] GrphEdge
```

162.4.3 Operations on Vertex-Sets and Edge-Sets

For each of the following operations, S and T may be interpreted as either the vertex-set or the edge-set of the graph G. The variable s may be interpreted as either a vertex or an edge. The edge-set and vertex-set support all the standard set operations.

`# S: GrphVertSet -> RngIntElt`

`# S: GrphEdgeSet -> RngIntElt`

The cardinality of the set S.

`s in S: GrphVert, GrphVertSet -> BoolElt`

`s in S: GrphEdge, GrphEdgeSet -> BoolElt`

Return `true` if the vertex (edge) s lies in the vertex-set (edge-set) S, otherwise `false`.

`s notin S: GrphVert, GrphVertSet -> BoolElt`

`s notin S: GrphEdge, GrphEdgeSet -> BoolElt`

Return `true` if the vertex (edge) s does not lie in the vertex-set (edge-set) S, otherwise `false`.

`S subset T: GrphVertSet, GrphVertSet -> BoolElt`

`S subset T: GrphEdgeSet, GrphEdgeSet -> BoolElt`

Return `true` if the vertex-set (edge-set) S is contained in the vertex-set (edge-set) T, otherwise `false`.

`S notsubset T: GrphVertSet, GrphVertSet -> BoolElt`

`S notsubset T: GrphEdgeSet, GrphEdgeSet -> BoolElt`

Return `true` if the vertex-set (edge-set) S is not contained in the vertex-set (edge-set) T, otherwise `false`.

`S eq T: GrphVertSet, GrphVertSet -> BoolElt`

`S eq T: GrphEdgeSet, GrphEdgeSet -> BoolElt`

Return `true` if the vertex-set (edge-set) S is equal to the vertex-set (edge-set) T.

`s eq t: GrphVert, GrphVert -> BoolElt`

`s eq t: GrphEdge, GrphEdge -> BoolElt`

Returns `true` if the vertex (edge) s is equal to the vertex (edge) t.

`S ne T: GrphVertSet, GrphVertSet -> BoolElt`

`S ne T: GrphEdgeSet, GrphEdgeSet -> BoolElt`

Returns `true` if the vertex-set (edge-set) S is not equal to the vertex-set (edge-set) T.

`s ne t: GrphVert, GrphVert -> BoolElt`

`s ne t: GrphEdge, GrphEdge -> BoolElt`

Returns `true` if the vertex (edge) s is not equal to the vertex (edge) t.

`ParentGraph(S): GrphVertSet -> Grph`

`ParentGraph(S): GrphEdgeSet -> Grph`

Return the graph G for which S is the vertex-set (edge-set).

`ParentGraph(s): GrphVert -> Grph`

`ParentGraph(s): GrphEdge -> Grph`

Return the graph G for which s is a vertex (edge).

`Random(S): GrphVertSet -> GrphVert`

`Random(S): GrphEdgeSet -> GrphEdge`

Choose a random element from the vertex-set (edge-set) S.

`Representative(S): GrphVertSet -> GrphVert`

`Rep(S): GrphVertSet -> GrphVert`

`Representative(S): GrphEdgeSet -> GrphEdge`

`Rep(S): GrphEdgeSet -> GrphEdge`

Choose some element from the vertex-set (edge-set) S.

`for x in S do ... end for;`

The vertex-set (edge-set) S may appear as the range in the `for`-statement.

`for random x in S do ... end for;`

The vertex-set (edge-set) S may appear as the range in the `for random` - statement.

162.4.4 Operations on Edges and Vertices

`EndVertices(e): GrphEdge -> { GrphVert}`

`EndVertices(e): GrphEdge -> [ GrphVert ]`

Given an edge e belonging to the graph G, return a set containing the two end-vertices of e. If G is a digraph return the two end-vertices in a sequence.

`InitialVertex(e): GrphEdge -> GrphVert`

Given an undirected or directed edge e from vertex u to vertex v, return vertex u. This is useful in the undirected case since it indicates, where relevant, the direction in which the edge has been traversed.

`TerminalVertex(e): GrphEdge -> GrphVert`

Given an undirected or directed edge e from vertex u to vertex v, return vertex v. This is useful in the undirected case since it indicates, where relevant, the direction in which the edge has been traversed.

`IncidentEdges(u): GrphVert -> { GrphEdge}`

Given a vertex u of a graph G, return the set of all edges incident with the vertex u. If G is directed, then the set consists of all the edges incident into u and from v.

162.5 Labelled, Capacitated, and Weighted Graphs

A vertex labelling of a graph G is a partial map f from the vertex-set V of G into a set L. An edge labelling of a graph G is a partial map f from the edge-set E of G into a set L. A labelled graph is built by assigning labels successively to vertices or edges after the (unlabelled) graph has been constructed.

Similarly, a capacitated graph G is a partial map from its edge-set into Z^+, and a weighted graph G is a partial map from its edge-set into R, R any ring with a total order. Those two last features are particularly convenient when running shortest-paths and flow algorithms. Edge capacities and edge weights are assigned to edges once the graph has been constructed. Any graph edge may carry a label, together with a capacity and/or a weight.

All the functions for decorating graph vertices and edges are fully documented in Section *Vertex and Edge Decorations* in Chapter *Multigraphs*. A few examples are also given there.

162.6 Standard Constructions for Graphs

The two main ways in which to construct a new graph from an old one are either by taking a subgraph of or by modifying the original graph. Another method is to take the quotient graph. When the first two methods are employed, the support set and vertex and edge decorations are retained in the resulting graph.

162.6.1 Subgraphs and Quotient Graphs

`sub< G | list >: Grph, List -> Grph, GrphVertSet, GrphEdgeSet`

Construct the graph H as a subgraph of G. The function returns three values: The graph H, the vertex-set V of H; and the edge-set E of H. If G has a support set and/or if G has vertex/edge decorations, then these attributes are transferred to the subgraph H.

The elements of V and of E are specified by the list *list* whose items can be objects of the following types:

(a) A vertex of G. The resulting subgraph will be the subgraph induced on the subset of `VertexSet(G)` defined by the vertices in *list*.

(b) An edge of G. The resulting subgraph will be the subgraph with vertex-set `VertexSet(G)` whose edge-set consists of the edges in *list*.

(c) A set of

(i) Vertices of G.

(ii) Edges of G.

It is easy to recover the map that maps the vertices of the subgraph to the vertices of the supergraph and vice-versa. We give an example below.

`quo< G | P >: Grph, {{ GrphVert}} -> Grph, GrphVertSet, GrphEdgeSet`

Given a graph G and a partition P of the vertex-set $V(G)$ of G, construct the quotient graph Q of G defined by P. The partition is given as a set of subsets of $V(G)$. Suppose the cells of the partition P are $P_1, \ldots, P_r$. The vertices of Q correspond to the cells P_i. Vertices v_i and v_j of Q are adjacent if and only if there is an edge in G joining a vertex in P_i with a vertex in P_j.

Example: Mapping Vertices of a Subgraph `(ex-8045cf)`

We start with a simple example which demonstrates how to map the vertices of a subgraph onto the vertices of the supergraph and vice-versa: This is achieved by simple coercion into the appropriate vertex-set.

```
> K5, V := CompleteGraph(5);
> K3, V1 := sub< K5 | { V | 3, 4, 5 } >;
> IsSubgraph(K5, K3);
true
>
> V!V1!1;
3
>
```

(continues on next page)

(continued from previous page)

```
> V1!V!4;
2
>
> V1!V!1;

>> V1!V!1;
     ^
Runtime error in '!': Illegal coercion
LHS: GrphVertSet
RHS: GrphVert
```

A 1-factor of a graph G is a set of disjoint edges which form a spanning forest for G. In the following example, we construct the graph that corresponds to K_6 with a 1-factor removed.

```
> K6, V, E := CompleteGraph(6);
> K6;
Graph
Vertex  Neighbours

1       2 3 4 5 6 ;
2       1 3 4 5 6 ;
3       1 2 4 5 6 ;
4       1 2 3 5 6 ;
5       1 2 3 4 6 ;
6       1 2 3 4 5 ;
> F1 := { E | {1,2}, {3, 4}, {5, 6} };
> G1, V1, E1 := K6 - F1;
> G1;
Graph
Vertex  Neighbours

1       3 4 5 6 ;
2       3 4 5 6 ;
3       1 2 5 6 ;
4       1 2 5 6 ;
5       1 2 3 4 ;
6       1 2 3 4 ;
```

Example: Quotient of a Complete Graph (ex-c8df39)

Taking the complete graph K_9, we form its quotient with respect to the partition of the vertex-set into three 3-sets.

```
> K9, V, E := CompleteGraph(9);
> P := { { V | 1, 2, 3}, { V | 4, 5, 6}, { V | 7, 8, 9} };
> Q := quo< K9 | P >;
> Q;
Graph
Vertex  Neighbours

1       2 3 ;
2       1 3 ;
3       1 2 ;
```

162.6.2 Incremental Construction of Graphs

Unless otherwise specified, each of the functions described in this section returns three values:

(i) The graph G;

(ii) The vertex-set V of G;

(iii) The edge-set E of G.

Adding Vertices

`G + n: Grph, RngIntElt -> Grph`

Given a graph G and a non-negative integer n, adds n new vertices to G. The existing vertex and edge decorations are retained, but the support will become the standard support.

`G +:= n: Grph, RngIntElt`

`AddVertex(~G): Grph`

`AddVertices(~G, n): Grph, RngIntElt`

The procedural version of the previous function. `AddVertex` adds one vertex only to G.

`AddVertex(~G, l): Grph, .`

Given a graph G and a label l, adds a new vertex with label l to G. The existing vertex and edge decorations are retained, but the support will become the standard support.

`AddVertices(~G, n, L): Grph, RngIntElt, SeqEnum`

Given a graph G and a non-negative integer n, and a sequence of n labels, adds n new vertices to G with labels from L. The existing vertex and edge decorations are retained, but the support will become the standard support.

Removing Vertices

`G - v: Grph, GrphVert -> Grph`

`G - U: Grph, { GrphVert } -> Grph`

Given a graph G and a vertex v or a set of vertices U of G, removes v or the vertices in U from G. The support and vertex and edge decorations are retained.

`G -:= v: Grph, GrphVert`

`G -:= U: Grph, { GrphVert }`

`RemoveVertex(~G, v): Grph, GrphVert`

`RemoveVertices(~G, U): Grph, { GrphVert }`

The procedural versions of the previous functions.

Adding Edges

`G + { u, v }: GrphUnd, { GrphVert, GrphVert } -> GrphUnd, GrphEdge`

`G + [ u, v ]: GrphDir, [ GrphVert, GrphVert ] -> GrphDir, GrphEdge`

Given a graph G and a pair of vertices of G, add the edge to G described by this pair. If G is undirected then the edge must be given as a set of (two) vertices, if G is directed the edge is given as a sequence of (two) vertices. The support and existing vertex and edge decorations are retained. This set of functions has two return values: The first is the modified graph and the second is the newly created edge.

`G + { { u, v } }: GrphUnd, { { GrphVert, GrphVert } } -> GrphUnd`

`G + { [ u, v ] }: GrphDir, { [ GrphVert, GrphVert ] } -> GrphDir`

Given a graph G and a set of pairs of vertices of G, add the edges to G described by these pairs. If G is undirected then edges must be given as a set of (two) vertices, if G is directed edges are given as a sequence of (two) vertices. The support and existing vertex and edge decorations are retained.

`G +:= { u, v }: GrphUnd, { GrphVert, GrphVert }`

`G +:= [ u, v ]: GrphDir, [ GrphVert, GrphVert ]`

`G +:= { { u, v } }: GrphUnd, { { GrphVert, GrphVert } }`

`G +:= { [ u, v ] }: GrphDir, { [ GrphVert, GrphVert ] }`

The procedural versions of the previous four functions.

`AddEdge(G, u, v): Grph, GrphVert, GrphVert -> Grph, GrphEdge`

Given a graph G, two vertices of G u and v, returns a new edge between u and v. The support and existing vertex and edge decorations are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

`AddEdge(G, u, v, l): Grph, GrphVert, GrphVert, . -> Grph, GrphEdge`

Given a graph G, two vertices of G u and v, and a label l, adds a new edge with label l between u and v. The support and existing vertex and edge decorations are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

`AddEdge(~G, u, v): Grph, GrphVert, GrphVert`

`AddEdge(~G, u, v, l): Grph, GrphVert, GrphVert, .`

Procedural versions of previous functions adding edges to a graph.

`AddEdges(G, S): GrphUnd, { { GrphVert, GrphVert } } -> GrphUnd`

`AddEdges(G, S): GrphDir, { [ GrphVert, GrphVert ] } -> GrphDir`

Given a graph G, a set S of pairs of vertices of G, adds the edges specified in S. The elements of S must be sets or sequences of two vertices of G, depending on whether G is undirected or directed respectively. The support and existing vertex and edge decorations are retained.

`AddEdges(G, S, L): Grph, SeqEnum, SeqEnum -> Grph`

Given a graph G, a sequence S of pairs of vertices of G, and a sequence L of labels of same length, adds the edges specified in S with its corresponding label in L. The elements of S must be sets or sequences of two vertices of G, depending on whether G is undirected or directed respectively. The support and existing vertex and edge decorations are retained.

`AddEdges(~G, S): GrphUnd, { { GrphVert, GrphVert } }`

`AddEdges(~G, S): GrphDir, { [ GrphVert, GrphVert ] }`

`AddEdges(~G, S, L): Grph, SeqEnum, SeqEnum`

Procedural versions of previous functions adding edges to a graph.

Removing Edges

`G - e: Grph, GrphEdge -> Grph`

`G - { e }: Grph, { GrphEdge } -> Grph`

Given a graph G and an edge e or a set S of edges of G, removes e or the edges in S from G. The resulting graph will have vertex-set $V(G)$ and edge-set $E(G) \setminus \{e\}$ or $E(G) \setminus S$. The support and existing vertex and edge decorations are retained.

`G - { { u, v } }: GrphUnd, { { GrphVert, GrphVert } } -> GrphUnd`

`G - { [u, v] }: GrphDir, { [ GrphVert, GrphVert ] } -> GrphDir`

Given a graph G and a set S of pairs $\{u, v\}$ or $[u, v]$, u, v vertices of G, form the graph having vertex-set $V(G)$ and edge-set $E(G)$ minus the edges from u to v for all pairs in S. An edge is represented as a set if G is undirected, as a set otherwise. The support and existing vertex and edge decorations are retained.

`G -:= e: Grph, GrphEdge`

`G -:= { e }: Grph, { GrphEdge }`

`G -:= { { u, v } }: GrphUnd, { { GrphVert, GrphVert rbrace }`

`G -:= { [u, v] }: GrphDir, { [ GrphVert, GrphVert ] }`

`RemoveEdge(~G, e): Grph, GrphEdge`

`RemoveEdges(~G, S): Grph, { GrphEdge }`

`RemoveEdge(~G, u, v): Grph, GrphVert, GrphVert`

`RemoveEdges(~G, S): GrphDir, { { GrphVert, GrphVert } }`

`RemoveEdges(~G, S): GrphDir, { [ GrphVert, GrphVert ] }`

The procedural versions of the previous functions.

162.6.3 Constructing Complements, Line Graphs; Contraction, Switching

Unless otherwise stated, each of the functions described here apply to both graphs and digraphs. Further, each of the functions returns three values:

(i) The graph G;

(ii) The vertex-set V of G;

(iii) The edge-set E of G.

`Complement(G): Grph -> Grph`

The complement of the graph G.

`Contract(e): GrphEdge -> Grph`

Given an edge $e = \{u, v\}$ of the graph G, form the graph obtained by identifying the vertices u and v, and removing any multiple edges (and loops if the graph is undirected) from the resulting graph. The graph's support and vertex/edges decorations are retained.

`Contract(u, v): GrphVert, GrphVert -> Grph`

Given vertices u and v belonging to the graph G, return the graph obtained by identifying the vertices u and v, and removing any multiple edges (and loops if the graph is undirected) from the resulting graph. The graph's support and vertex/edges decorations are retained.

`Contract(S): { GrphVert} -> Grph`

Given a set S of vertices belonging to the graph G, return the graph obtained by identifying all of the vertices in S, and removing any multiple edges (and loops if the graph is undirected) from the resulting graph. The graph's support and vertex/edges decorations are retained.

`InsertVertex(e): GrphEdge -> Grph`

Given an edge e in the graph G, insert a new vertex of degree 2 in e. If applicable, the two new edges thus created that replace e will have the same capacity and weight as e. They will be unlabelled. The vertex labels and the edge decorations of G are retained, but the resulting graph will have standard support.

`InsertVertex(T): { GrphEdge} -> Grph`

Given an a set T of edges belonging to the graph G, insert a vertex of degree 2 in each edge belonging to the set T. Vertex and edge decorations are handled as in the single edge case.

`LineGraph(G): Grph -> Grph`

The line graph of the non-empty graph G.

`Switch(u): GrphVert -> GrphUnd`

Given a vertex u in an undirected graph G, construct an undirected graph H from G, where H has the same vertex-set as G and the same edge-set, except that the vertices that were the neighbours of u in G become the non-neighbours of u in H, and the vertices that were non-neighbours of u in G become the neighbours of u in H. The support and vertex labels are *not* retained in the resulting graph.

`Switch(S): { GrphVert} -> Grph`

Given a set S of vertices belonging to the undirected graph G, apply the function `Switch(u)`, to each vertex of S in turn. The support and vertex labels are *not* retained in the resulting graph.

Example: Constructing the Grotzsch Graph `(ex-be173e)`

We illustrate the use of some of these operations by using them to construct the Grötzch graph. This graph may be built by taking the complete graph $K5$, choosing a cycle of length 5 (say,

1-3-5-2-4), inserting a vertex of degree two on each edge of this cycle and finally connecting each of these vertices to a new vertex.

```
> G := CompleteGraph(5);
> E := EdgeSet(G);
> H := InsertVertex({ E | { 1, 3 }, { 1, 4 }, { 2, 4 }, { 2, 5 }, { 3, 5 } } );
> L := Union(H, CompleteGraph(1));
> V := VertexSet(L);
> L := L + {  { V.11, V.6 }, { V.11, V.7 }, { V.11, V.8 }, { V.11, V.9 },
>            { V.11, V.10 } };
> L;
Graph
Vertex  Neighbours

1       2 5 6 7 ;
2       1 3 8 10 ;
3       2 4 6 9 ;
4       3 5 7 8 ;
5       1 4 9 10 ;
6       1 3 11 ;
7       1 4 11 ;
8       2 4 11 ;
9       3 5 11 ;
10      2 5 11 ;
11      6 7 8 9 10 ;
```

162.7 Unions and Products of Graphs

The support and vertex/edge decorations of the original graphs are *not* retained in the graph resulting from applying any of the union functions below.

Union(G, H): GrphUnd, GrphUnd -> GrphUnd

Union(G, H): GrphDir, GrphDir -> GrphDir

G join H: GrphDir, GrphDir -> GrphDir

G join H: GrphUnd, GrphUnd -> GrphUnd

Given graphs G and H with disjoint vertex sets $V(G)$ and $V(H)$, respectively, construct their union, i.e. the graph with vertex-set $V(G) \cup V(H)$, and edge-set $E(G) \cup E(H)$.

EdgeUnion(G, H): GrphDir, GrphDir -> GrphDir

EdgeUnion(G, H): GrphUnd, GrphUnd -> GrphUnd

Given graphs G and H having the same number of vertices, construct their edge union K. This construction identifies the i-th vertex of G with the i-th vertex of H for all i. The edge union has the same vertex-set as G (and hence as H) and vertices u and v of K are adjacent if and only if either u and v are adjacent in G or u and v are adjacent in H.

CompleteUnion(G, H): GrphDir, GrphDir -> GrphDir

CompleteUnion(G, H): GrphUnd, GrphUnd -> GrphUnd

Given graphs G and H with disjoint vertex-sets $V(G)$ and $V(H)$, respectively, construct the complete union of G and H. This graph consists of the union of G and H (**Union(G, H)**), together with edges uv, for all u in $V(G)$ and all v in $V(H)$.

CartesianProduct(G, H): GrphDir, GrphDir -> GrphDir

CartesianProduct(G, H): GrphUnd, GrphUnd -> GrphUnd

Given graphs G and H with disjoint vertex-sets $V(G)$ and $V(H)$, respectively, form the product $K = G \times H$ of G and H. The product has vertex-set $V(G) \times V(H)$. Two vertices $u = (u_1, u_2)$ and $v = (v_1, v_2)$ of K are adjacent when either

(a) $u_1 = v_1$ and u_2 adjv_2, or

(b) $u_2 = v_2$ and u_1 adjv_1.

`LexProduct(G, H): GrphDir, GrphDir -> GrphDir`

`LexProduct(G, H): GrphUnd, GrphUnd -> GrphUnd`

Given graphs G and H with disjoint vertex-sets $V(G)$ and $V(H)$, respectively, form the lexicographic product K of G and H. The lexicographic product has vertex-set $V(G) \times V(H)$. Two vertices $u = (u_1, u_2)$ and $v = (v_1, v_2)$ of K are adjacent when either

(a) u_1 adjv_1, or

(b) $u_1 = v_1$ and u_2 adjv_2.

`TensorProduct(G, H): GrphDir, GrphDir -> GrphDir`

`TensorProduct(G, H): GrphUnd, GrphUnd -> GrphUnd`

Given graphs G and H with disjoint vertex-sets $V(G)$ and $V(H)$, respectively, form the tensor product K of G and H. This graph has vertex-set $V(G) \times V(H)$. Two vertices $u = (u_1, u_2)$ and $v = (v_1, v_2)$ of K are adjacent when u_1 adjv_1 and u_2 adjv_2.

`G ^ n: GrphUnd, RngIntElt -> GrphUnd`

Given a graph G and a positive integer n, construct the n-th power K of G. This graph has the same vertex-set as G, and vertices u and v of K are adjacent if and only if the distance between u and v in G is less than or equal to n.

162.8 Converting between Graphs and Digraphs

Note that the two functions `UnderlyingGraph` and `UnderlyingDigraph` may also be used when one needs to get a copy of a graph G without G's support and vertex/edge decorations.

`OrientatedGraph(G): GrphUnd -> GrphDir`

Given a graph G, produce a digraph D whose vertex-set is the same as that of G and whose edge-set consists of the edges of G, each given a direction. The edges of D are always directed from the lower numbered vertex to the higher numbered vertex. Thus, if G contains the edge $\{u, v\}$, then D will have the edge $[u, v]$ if $u < v$, otherwise the edge $[u, v]$. The support and vertex/edge decorations of G are not retained.

`UnderlyingGraph(D): Grph -> GrphUnd`

The underlying graph G of the graph D; G has the same vertex-set as D. If D is undirected, then G is a copy of D without D's support and vertex/edge decorations. If D is directed, then two vertices u and v are adjacent in G if and only if, in D, there is either an edge directed from u to v or from v to u. The support and vertex/edge decorations of G are not retained.

`UnderlyingDigraph(G): Grph -> GrphDir`

The underlying digraph D of the graph G; D has the same vertex-set as G. If G is directed, then D is a copy of G without D's support and vertex/edge decorations. If G is undirected, then if vertices u and v are adjacent in G then, in D, there will be both an edge directed from u to v and an edge directed from v to u. The support and vertex/edge decorations of G are not retained.

162.9 Construction from Groups, Codes, and Designs

162.9.1 Graphs Constructed from Groups

`CayleyGraph(A : parameter): Grp -> Grph, GrphVertSet, GrphEdgeSet`

```
Labelled: BoolElt                    Default: true
Directed: BoolElt                    Default: true
```

Given a finite group A defined on generating set X, construct the Cayley graph C of A relative to the generating set X. This graph is defined as follows: The vertices correspond to the elements of A and two vertices u, v are adjacent if and only if there exists an element x in X such that $u * x = v$.

The optional parameter `Labelled` (`Labelled := true` by default) can be set to `false` to prevent the graph being labelled. If this is not done, then the vertices of C will be labelled with the appropriate elements of A and the (directed) edge from u to v will be labelled with the appropriate element x as defined above.

The parameter `Directed` (`Directed := true` by default) may be used to return the Cayley graph of G as an undirected graph

`SchreierGraph(A, B): Grp, Grp -> Grph, GrphVertSet, GrphEdgeSet`

```
Labelled: BoolElt                    Default: true
Directed: BoolElt                    Default: true
```

Given a finite group A defined on the generating set X and a subgroup B of A, construct the Schreier coset graph S for A over B, relative to X. The graph S is defined as follows: The vertices correspond to the cosets of B in A, and two vertices u, v are adjacent in S if and only if there exists an element x in X such that $u * x = v$.

The graph is available in both a labelled and an unlabelled version and directed and undirected versions. These versions are controlled by the parameters `Labelled` and `Directed`, which are both `true` by default.

`OrbitalGraph(P, u, T): GrpPerm, RngIntElt, { RngIntElt} -> GrphUnd`

Let P be a transitive permutation group acting on the set $\Omega = \{1, \ldots, n\}$. Let u be an element of Ω and let $T = \{t_1, \ldots, t_r\}$ be a subset of Ω. This function constructs the underlying graph G of the digraph corresponding to the union of P-orbits containing the pairs $(u, t_1), \ldots, (u, t_r)$. Thus, if T defines a self-paired orbit Δ of the stabilizer in P of the point u, this function constructs the orbital graph associated with Δ.

`ClosureGraph(P, G): GrpPerm, GrphUnd -> GrphUnd`

Let P be a permutation group acting on the set $\Omega = \{1, \ldots, n\}$. Let G be a graph (digraph) with vertices $v_1, \ldots, v_n$. This function adds the minimum number of edges to G so as to produce a graph (digraph) H which is left invariant by the group P.

`PaleyGraph(q): RngIntElt -> GrphUnd`

The Paley graph of $\mathbf{F}_q$ where q must be a prime power equivalent to 1 mod 4. Vertices are in bijection with elements of $\mathbf{F}_q$ and distinct elements are adjacent when their difference is a square in the field.

`PaleyTournament(q): RngIntElt -> GrphDir`

The Paley tournament of $\mathbf{F}_q$ where q must be a prime power equivalent to 3 mod 4. Vertices are in bijection with elements of $\mathbf{F}_q$ and there is an edge from u to v when $u \neq v$ and $v - u$ is a square in the field.

162.9.2 Graphs Constructed from Designs

`IncidenceGraph(D): Inc -> GrphUnd`

Given an incidence structure $D = (X, \mathcal{B})$, construct the incidence graph G of D. The vertices of G is $X \cup \mathcal{B}$. The adjacency rules are as follows: No two vertices of X are adjacent; no two vertices of $\mathcal{B}$ are adjacent; a vertex $x \in X$ is adjacent to a vertex $B \in \mathcal{B}$ if and only if $x \in B$.

`PointGraph(D): Inc -> GrphUnd`

Given an incidence structure $D = (X, \mathcal{B})$, construct the point graph G of D. The vertex-set of G is X. Vertices $x \in X$, $y \in X$ are adjacent in G if there is a block $B \in \mathcal{B}$ such that $x \in B$ and $y \in B$.

`BlockGraph(D): Inc -> GrphUnd`

The block graph of the incidence structure D; i.e. the point graph of the dual of D.

`IncidenceGraph(P): Plane -> GrphUnd`

Given a plane P with point-set V and line-set L, construct the incidence graph G of P. The vertex-set of G is $V \cup L$. The adjacency rules are as follows: No two vertices of V are adjacent; no two vertices of L are adjacent; a vertex $v \in V$ is adjacent to a vertex $a \in L$ if and only if v lies on a.

`PointGraph(P): Plane -> GrphUnd;`

Given a plane P with point-set V and line-set L, construct the point graph G of P. The vertex-set of G is V. Vertices $u, v \in V$ are adjacent in G iff there is a line in L that contains them both.

`LineGraph(P): Plane -> GrphUnd`

Given a plane P with point-set V and line-set L, construct the line graph G of P. The vertex-set of G is L. Lines $a, b \in L$ are adjacent in G iff there is a vertex in V that lies on them both.

`HadamardGraph(H : parameters): Mtrx -> GrphUnd`

Labels: BoolElt	Default: false

The graph of the ± 1 matrix H as described in Brendan D. McKay's note "Hadamard equivalence via graph isomorphism" (with self-loops omitted). The parameter `Labels` is set to `false` by default, but when set to `true`, the vertices associated with rows are labelled "row" and the others "col". Those labelled "row" are those given loops in McKay's paper.

162.9.3 Miscellaneous Graph Constructions

`Converse(G): GrphDir -> GrphDir`

Returns the converse H of the directed graph G: if $[u, v]$ is an edge of G then $[v, u]$ is an edge of H.

`OddGraph(n): RngIntElt -> GrphUnd`

The nth odd graph. Vertices are $(n-1)$–subsets of a $(2n-1)$-set with vertices adjacent if and only if the $(n-1)$–subsets are disjoint.

`TriangularGraph(n): RngIntElt -> GrphUnd`

The nth triangular graph. Vertices are 2-subsets of a n-set with vertices adjacent if and only if the 2-subsets are unequal and not disjoint.

`SquareLatticeGraph(n): RngIntElt -> GrphUnd`

The nth square lattice graph. This is the cartesian product of the nth complete graph with itself.

`ClebschGraph() -> GrphUnd`

`ShrikhandeGraph() -> GrphUnd`

`GewirtzGraph() -> GrphUnd`

`PetersenGraph() -> GrphUnd`

Return the named graph.

`ChangGraphs() -> [GrpUnd, GrpUnd, GrpUnd]`

Return a sequence of the three Chang graphs.

162.10 Elementary Invariants of a Graph

`Order(G): Grph -> RngIntElt`

`NumberOfVertices(G): Grph -> RngIntElt`

The number of vertices of the graph G.

`Size(G): Grph -> RngIntElt`

`NumberOfEdges(G): Grph -> RngIntElt`
The number of edges of the graph G.

`CharacteristicPolynomial(G): GrphUnd -> RngUPolElt`
The characteristic polynomial (over the integers) of the graph G; i.e. the characteristic polynomial of the adjacency matrix of G.

`Spectrum(G): GrphUnd -> SetEnum`
The spectrum of the graph G; i.e. the roots of the characteristic polynomial of G. The roots are returned as a set of tuples, each containing a root and its multiplicity.

162.11 Elementary Graph Predicates

Let G and H be two graphs. For clarity, we list here the conditions under which G is equal to H and H is a subgraph of G. The conditions take into account the fact that the graphs may have a support and that their vertex and edges may be decorated.

The graphs G and H are equal if and only if:

- they are of the same type,
- they are structurally identical,
- they have the same support,
- they have identical vertex and edge labels,
- if applicable, the total capacity from u to v in G is equal to the total capacity from u to v in H.

Also, H is a subgraph of G if and only if:

- they are of the same type,
- H is a structural subgraph of G,
- any vertex `v` in H has the same support as the vertex `VertexSet(G)!v` in G,
- any vertex `v` in H has the same label as the vertex `VertexSet(G)!v` in G,
- any edge `e` in H has the same label as the edge `EdgeSet(G)!e` in G,
- if applicable, the total capacity from u to v in G is at least as large as the total capacity from u to v in H.

Note that the truth value of the above two tests is not dependent on the weights of the edges of the graphs, should these edges be weighted. One could argue that given this shortcoming, the tests should not be dependent on the capacities of the edges either. We welcome users' comments on this matter.

`u adj v: GrphVert, GrphVert -> BoolElt`
Let u and v be two vertices of the same graph G. If G is undirected, returns `true` if and only if u and v are adjacent. If G is directed, returns `true` if and only if there is an edge directed from u to v.

`e adj f: GrphEdge, GrphEdge -> BoolElt`
Let e and f be two edges of the same graph G. If G is undirected, returns `true` if and only if e and f share a common vertex. If G is directed, returns `true` if and only if the terminal vertex of e (f) is the initial vertex of f (e).

`u notadj v: GrphVert, GrphVert -> BoolElt`
The negation of the `adj` predicate applied to vertices.

`e notadj f: GrphEdge, GrphEdge -> BoolElt`
The negation of the `adj` predicate applied to edges.

`u in e: GrphVert, GrphEdge -> BoolElt`
Let u be a vertex and e an edge of a graph G. Returns `true` if and only if u is an end-vertex of e.

`u notin e: GrphVert, GrphEdge -> BoolElt`

The negation of the `in` predicate applied to a vertex with respect to an edge.

`G eq H: GrphDir, GrphDir -> BoolElt`

`G eq H: GrphUnd, GrphUnd -> BoolElt`

Returns `true` if the graphs G and H are identical, otherwise `false`. G and H are identical if and only if:

- they are structurally identical,
- they have the same support,
- they have identical vertex and edge labels,
- if applicable, the total capacity from u to v in G is equal to the total capacity from u to v in H.

Thus, as an example, if G and H are structurally identical graphs, and the vertices of G are labelled while the vertices of H are not, then `G eq H` returns `false`.

`IsSubgraph(G, H): Grph, Grph -> BoolElt`

Returns `true` if H is a subgraph of G, otherwise `false`. H is a subgraph of G if and only if:

- H can be determined to be a structural subgraph of G,
- any vertex `v` in H has the same support as the vertex `VertexSet(G)!v` in G,
- any vertex `v` in H has the same label as the vertex `VertexSet(G)!v` in G,
- any edge `e` in H has the same label as the edge `EdgeSet(G)!e` in G.
- if applicable, the total capacity from u to v in G is at least as large as the total capacity from u to v in H.

`IsBipartite(G): GrphUnd -> BoolElt`

Returns `true` if the graph G is a bipartite graph, otherwise `false`.

`IsComplete(G): Grph -> BoolElt`

Returns `true` if the graph G on n vertices is the complete graph on n vertices, otherwise `false`.

`IsEulerian(G): Grph -> BoolElt`

Returns `true` if the graph G is an eulerian graph, otherwise `false`. A eulerian graph is a graph whose vertices have all even degree. An eulerian digraph is a digraph whose vertices have same in and outdegree. That is, if D is a digraph, D is eulerian if and only if `OutDegree(v)` equals `InDegree(v)` for all vertices v of D.

`IsForest(G): GrphUnd -> BoolElt`

Returns `true` if the graph G is a forest, i.e. does not possess any cycles, otherwise `false`.

`IsEmpty(G): Grph -> BoolElt`

Returns `true` if the graph G is an empty graph, otherwise `false`. A graph is empty if its edge-set E is empty.

`IsNull(G): Grph -> BoolElt`

Returns `true` if the graph G is a null graph, otherwise `false`. A graph is null if its vertex-set V is empty.

`IsPath(G): Grph -> BoolElt`

Returns `true` if the graph G is a path graph, otherwise `false`.

`IsPolygon(G): Grph -> BoolElt`

Returns `true` if the graph G is a polygon graph, otherwise `false`.

`IsRegular(G): Grph -> BoolElt`

Returns `true` if the graph G is a regular graph, otherwise `false`.

`IsTree(G): Grph -> BoolElt`

Returns `true` if the graph G is a tree, otherwise `false`.

162.12 Adjacency and Degree

162.12.1 Adjacency and Degree Functions for a Graph

`Degree(u): GrphVert -> RngIntElt`

Given a vertex u of the graph G, return the degree of u, ie the number of edges incident to u.

`Alldeg(G, n): GrphUnd, RngIntElt -> { GrphVert}`

Given a graph G, and a non-negative integer n, return the set of all vertices of G that have degree equal to n.

`MaximumDegree(G): GrphUnd -> RngIntElt, GrphVert`

`Maxdeg(G): GrphUnd -> RngIntElt, GrphVert`

The maximum of the degrees of the vertices of the graph G. This function returns two values: the maximum degree, and a vertex of G having that degree.

`MinimumDegree(G): GrphUnd -> RngIntElt, GrphVert`

`Mindeg(G): GrphUnd -> RngIntElt, GrphVert`

The minimum of the degrees of the vertices of the graph G. This function returns two values: the minimum degree, and a vertex of G having that degree.

`DegreeSequence(G): Grph -> [ { GrphVert} ]`

Given a graph G such that the maximum degree of any vertex of G is r, return a sequence D of length $r + 1$, such that $D[i]$, $1 \leq i \leq r + 1$, is the number of vertices in G having degree $i - 1$.

`Valence(G): GrphUnd -> RngIntElt`

Given a regular graph G, return the valence of G (the degree of any vertex).

`Neighbours(u): GrphVert -> { GrphVert}`

`Neighbors(u): GrphVert -> { GrphVert}`

Given a vertex u of the graph G, return the set of vertices of G that are adjacent to u.

`IncidentEdges(u): GrphVert -> { GrphEdge}`

The set of all edges incident with the vertex u.

`Bipartition(G): GrphUnd -> [ { GrphVert} ]`

Given a bipartite graph G, return its two partite sets in the form of a pair of subsets of $V(G)$.

`MinimumDominatingSet(G): GrphUnd -> SetEnum`

A dominating set S of a graph G is such that the vertices of S together with the vertices adjacent to vertices in S form the vertex-set of G. A dominating set S is minimal if no proper subset of S is a dominating set. A minimum dominating set is a minimal dominating set of smallest size. The algorithm implemented is a backtrack algorithm (see [Chr75] p. 41).

162.12.2 Adjacency and Degree Functions for a Digraph

`InDegree(u): GrphVert -> RngIntElt`

The number of edges directed into the vertex u belonging to the directed graph G.

`OutDegree(u): GrphVert -> RngIntElt`

The number of edges of the form uv where u is a vertex belonging to the directed graph G.

`Degree(u): GrphVert -> RngIntElt`

Given a vertex u belonging to the digraph G, return the total degree of u, i.e. the sum of the in–degree and out–degree for u.

`Alldeg(G, n): GrphDir, RngIntElt -> { GrphVert}`

Given a digraph G, and a non–negative integer n, return the set of all vertices of G that have total degree equal to n.

`MaximumInDegree(G): GrphDir -> RngIntElt, GrphVert`

`Maxindeg(G): GrphDir -> RngIntElt, GrphVert`

The maximum indegree of the vertices of the digraph G. This function returns two values: the maximum indegree, and the first vertex of G having that degree.

`MaximumOutDegree(G): GrphDir -> RngIntElt, GrphVert`

`Maxoutdeg(G): GrphDir -> RngIntElt, GrphVert`

The maximum outdegree of the vertices of the digraph G. This function returns two values: the maximum outdegree, and the first vertex of G having that degree.

`MinimumInDegree(G): GrphDir -> RngIntElt, GrphVert`

`Minindeg(G): GrphDir -> RngIntElt, GrphVert`

The minimum indegree of the vertices of the digraph G. This function returns two values: the minimum indegree, and the first vertex of G having that degree.

`MinimumOutDegree(G): GrphDir -> RngIntElt, GrphVert`

`Minoutdeg(G): GrphDir -> RngIntElt, GrphVert`

The minimum outdegree of the vertices of the digraph G. This function returns two values: the minimum outdegree, and the first vertex of G having that degree.

`MaximumDegree(G): GrphDir -> RngIntElt, GrphVert`

`Maxdeg(G): GrphDir -> RngIntElt, GrphVert`

The maximum total degree of the vertices of the digraph G. This function returns two values: the maximum total degree, and the first vertex of G having that degree.

`MinimumDegree(G): GrphDir -> RngIntElt, GrphVert`

`Mindeg(G): GrphDir -> RngIntElt, GrphVert`

The minimum total degree of the vertices of the digraph G. This function returns two values: the minimum total degree, and the first vertex of G having that degree.

`DegreeSequence(G): Grph -> [ { GrphVert} ]`

Given a digraph G such that the maximum degree of any vertex of G is r, return a sequence D of length $r + 1$, such that $D[i]$, $1 \leq i \leq r + 1$, is the number of vertices in G having degree $i - 1$.

`InNeighbours(u): GrphVert -> { GrphVert}`

`InNeighbors(u): GrphVert -> { GrphVert}`

Given a vertex u of the digraph G, return the set containing all vertices v such that vu is an edge in the digraph, i.e. the starting points of all edges that are directed into the vertex u.

`OutNeighbours(u): GrphVert -> { GrphVert}`

`OutNeighbors(u): GrphVert -> { GrphVert}`

Given a vertex u of the digraph G, return the set of vertices v of G such that uv is an edge in the graph G, i.e. the set of vertices v that are the end vertices of edges directed from u to v.

`IncidentEdges(u): GrphVert -> { GrphEdge}`

The set of all edges incident with the vertex u.

162.13 Connectedness

162.13.1 Connectedness in a Graph

`IsConnected(G): GrphUnd -> BoolElt`

Returns `true` if and only if the graph G is connected.

`Components(G): GrphUnd -> [ { GrphVert} ]`

The connected components of the graph G. These are returned in the form of a sequence of subsets of the vertex-set of G.

`Component(u): GrphVert -> Grph`

The subgraph corresponding to the connected component of the graph G containing vertex u. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

`IsSeparable(G): GrphUnd -> BoolElt`

Returns `true` if and only if the graph G is connected and has at least one cut vertex.

`IsBiconnected(G): GrphUnd -> BoolElt`

Returns `true` if and only if the graph G is biconnected.

`CutVertices(G): Grph -> { GrphVert}`

The set of cut vertices for the connected graph G (a set of vertices).

`Bicomponents(G): GrphUnd -> [GrphUnd]`

The biconnected components of the graph G. These are returned in the form of a sequence of subsets of the vertex-set of G. The graph may be disconnected.

162.13.2 Connectedness in a Digraph

`IsStronglyConnected(G): GrphDir -> BoolElt`

Returns `true` if and only if the digraph G is strongly connected.

`IsWeaklyConnected(G): GrphDir -> BoolElt`

Returns `true` if and only if the digraph G is weakly connected.

`StronglyConnectedComponents(G): GrphDir -> [ { GrphVert} ]`

The strongly connected components of the digraph G. These are returned in the form of a sequence of subsets of the vertex-set of G.

`Component(u): GrphVert -> Grph`

The subgraph corresponding to the connected component of the digraph G containing vertex u. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

162.13.3 Graph Triconnectivity

The linear-time triconnectivity algorithm by Hopcroft and Tarjan [HT73] has been implemented with corrections of our own and from C. Gutwenger and P. Mutzel [GM01]. This algorithm requires that the graph has a sparse representation. The input graph may be disconnected or have cut vertices.

The algorithm completely splits the graph into components that can later be reconstructed as triconnected graphs: It is our belief that the word "tricomponents" is misused in all the papers discussing Hopcroft and Tarjan's algorithm, including [HT73] and [GM01]. Let us clarify this point.

Assume that G is a biconnected but not triconnected graph with vertex-set V and with one separation pair (a, b) which splits $V - \{a, b\}$ into V_1 and V_2. That is, for any vertices $u \in V_1$ and $v \in V_2$, every path from u to v in G contains at least one of a or b.

Now, the algorithm will correctly find the separation pair (a, b) and return the two subsets of V, $V_1 + \{a, b\}$ and $V_2 + \{a, b\}$. Let G_1 and G_2 be the subgraphs of G induced by $V_1 + \{a, b\}$ and $V_2 + \{a, b\}$ respectively. We say that the algorithm *splits* G into G_1 and G_2.

But G_1 and G_2 are triconnected if and only if there is an edge (a, b) in G. We call G_1 and G_2 the *split components* of G.

More generally, if $G_1, \ldots, G_m$ are the split components of a graph G, one can make the G_i triconnected by adding the edges (a, b) to them for all the separation pairs (a, b) of G with a, $b \in G_i$ for some i.

The algorithm finds all the cut vertices and/or the separation pairs of a graph, unless only the boolean test is applied. When this is the case the algorithm stops as soon as a cut vertex or separation pair is found.

The triconnectivity algorithm only applies to undirected graphs.

`IsTriconnected(G): GrphUnd -> BoolElt`

Returns `true` if and only if the graph G is triconnected.

`Splitcomponents(G): GrphUnd -> [ { GrphVert } ], [ [ GrphVert ]]`

The split components of the graph G. These are returned in the form of a sequence of subsets of the vertex-set of G. The graph may be disconnected. The second return value returns the cut vertices and the separation pairs.

`SeparationVertices(G): GrphUnd -> [ [ GrphVert ]], [ { GrphVert} ]`

The cut vertices and/or the separation pairs of the graph G as a sequence of vertices or pairs of vertices of G. The graph may be disconnected. The second return value returns G's split components.

Example: Triconnectivity of a Graph (ex-1455c3)

```
> G := Graph< 11 |
> {1, 3}, {1, 4}, {1, 7}, {1, 11}, {1, 2},
> {2, 4}, {2, 3},
> {3, 4},
> {4, 7}, {4, 11}, {4, 7}, {4, 5},
> {5, 10}, {5, 9}, {5, 8}, {5, 6},
> {6, 8}, {6, 7},
> {7, 8},
> {9, 11}, {9, 10},
> {10, 11}: SparseRep := true >;
> TS, PS := Splitcomponents(G);
> TS;
[
  { 5, 6, 7, 8 },
  { 5, 9, 10, 11 },
  { 1, 4, 5, 7, 11 },
  { 1, 4 },
  { 1, 2, 3, 4 }
]
> PS;
[
  [ 5, 7 ],
  [ 5, 11 ],
  [ 1, 4 ]
]
```

Let `G1` be the subgraph induced by `TS[1]`. We will verify that it is not triconnected and we make it triconnected by adding the edge `[5, 7]`, since `[5, 7]` is a separation pair of `G` whose vertices belong to `G1`.

```
> G1 := sub< G | TS[1] >;
> IsTriconnected(G1);
false
> AddEdge(~G1, Index(VertexSet(G1)!VertexSet(G)!5),
>       Index(VertexSet(G1)!VertexSet(G)!7));
> IsTriconnected(G1);
true
```

162.13.4 Maximum Matching in Bipartite Graphs

The maximum matching algorithm is a flow-based algorithm. We have implemented two maximum flow algorithms, the Dinic algorithm, and a push-relabel algorithm. The latter is almost always the most efficient, it may be outperformed by Dinic only in the case of very sparse graphs. For a full discussion of these algorithms, the reader is referred to Section *Maximum Flow and Minimum Cut* in Chapter *Multigraphs*. Example *Bipartite Matching via Maximum Flow* in this same chapter actually is an implementation in Magma code of the maximum matching algorithm.

MaximumMatching(G): GrphUnd -> [{ GrphEdge}]

```
Al: MonStgElt                    Default: "PushRelabel"
```

A maximum matching in the bipartite graph G. The matching is returned as a sequence of edges of G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

Example: Maximum Matching in a Bipartite Graph (ex-251f2b)

We begin by creating a random bipartite graph. The matching algorithm is flow-based and is one of the algorithms which deals with graphs as adjacency lists. For this reason we might as well create the graph with a sparse representation (see Section *Graphs with a Sparse Representation*).

```
> n := 5;
> m := 7;
> d := 3;
> G := EmptyGraph(n);
> H := EmptyGraph(m);
> G := G join H;
> for u in [1..n] do
>     t := Random(0, d);
>     for tt in [1..t] do
>         v := Random(n+1, n+m);
>         AddEdge(~G, u, v);
>     end for;
> end for;
> G := Graph< Order(G) | G : SparseRep := true >;
>
> IsBipartite(G);
true
> Bipartition(G);
[
  { 1, 12, 2, 3, 4, 5, 7 },
  { 11, 6, 8, 9, 10 }
]
> MaximumMatching(G);
[ {1, 11}, {4, 10}, {2, 9}, {5, 8}, {3, 6} ]
```

162.13.5 General Vertex and Edge Connectivity in Graphs and Digraphs

As for the maximum matching algorithm (Subsection *Maximum Matching in Bipartite Graphs*) the vertex and edge connectivity algorithms are flow-based algorithms. We have implemented two flow algorithms, Dinic and a push relabel method. In general Dinic outperforms the push relabel method only in the case of very sparse graphs. This is particularly visible when computing the edge connectivity of a graph. For full details on those flow algorithms, refer to Section *Maximum Flow and Minimum Cut* in Chapter *Multigraphs*. For details on how vertex and edge connectivity are implemented, see [Eve79].

The general connectivity algorithms apply to both undirected and directed graphs.

VertexSeparator(G): Grph -> [GrphVert]

```
Al: MonStgElt                    Default: "PushRelabel"
```

If G is an undirected graph, a *vertex separator* for G is a smallest set of vertices S such that for any u, $v \in V(G)$, every path connecting u and v passes through at least one vertex of S.

If G is a directed graph, a vertex separator for G is a smallest set of vertices S such that for any u, $v \in V(G)$, every directed path from u to v passes through at least one vertex of S.

`VertexSeparator` returns the vertex separator of G as a sequence of vertices of G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`VertexConnectivity(G): Grph -> RngIntElt, [ GrphVert ]`

```
Al: MonStgElt                    Default: "PushRelabel"
```

Returns the vertex connectivity of the graph G, the size of a minimum vertex separator of G. Also returns a vertex separator for G as the second value. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`IsKVertexConnected(G, k): Grph, RngIntElt -> BoolElt`

```
Al: MonStgElt                    Default: "PushRelabel"
```

Returns `true` if the graph G's vertex connectivity is at least k, `false` otherwise. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`EdgeSeparator(G): Grph -> [ GrphEdge ]`

```
Al: MonStgElt                    Default: "PushRelabel"
```

If G is an undirected graph, an *edge separator* for G is a smallest set of edges T such that for any u, $v \in V(G)$, every path connecting u and v passes through at least one edge of T.

If G is a directed graph, an edge separator for G is a smallest set of edges T such that for any u, $v \in V(G)$, every directed path from u to v passes through at least one edge of T.

`EdgeSeparator` returns the edge separator of G as a sequence of edges of G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`EdgeConnectivity(G): Grph -> RngIntElt, [ GrphEdge ]`

```
Al: MonStgElt                    Default: "PushRelabel"
```

Returns the edge connectivity of the graph G, the size of a minimum edge separator of G. Also returns as the second value an edge separator for G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`IsKEdgeConnected(G, k): Grph, RngIntElt -> BoolElt`

```
Al: MonStgElt                    Default: "PushRelabel"
```

Returns `true` if the graph G's edge connectivity is at least k, `false` otherwise. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

Example: Vertex and Edge Connectivity `(ex-c02f9d)`

We compute the vertex and edge connectivity for a small undirected graph and we perform a few checks. The connectivity algorithm is one of the algorithms which deals with graphs as adjacency

lists. For this reason we might as well create the graph with a sparse representation (see Section *Graphs with a Sparse Representation*).

```
> G := Graph< 4 | {1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 4}
> : SparseRep := true >;
> IsConnected(G);
true
>
> vc := VertexConnectivity(G);
> S := VertexSeparator(G);
> vc;
2
> S;
[ 3, 2 ]
>
> H := G;
> vs := [ Integers() | Index(x) : x in S ];
> RemoveVertices(~H, vs);
> IsConnected(H);
false
> for k in [0..vc] do
>     IsKVertexConnected(G, k);
> end for;
true
true
true
> for k in [vc+1..Order(G)-1] do
>     IsKVertexConnected(G, k);
> end for;
false
>
>
>
> ec := EdgeConnectivity(G);
> T :=  EdgeSeparator(G);
> ec;
2
> T;
[ {1, 2}, {1, 3} ]
>
> H := G;
> M := [ { Index(e[1]), Index(e[2]) }
> where e := SetToSequence(EndVertices(e)) : e in T ];
> RemoveEdges(~H, M);
> IsConnected(H);
false
> for k in [0..ec] do
>     IsKEdgeConnected(G, k);
> end for;
true
true
true
> for k in [ec+1..Order(G)-1] do
>     IsKEdgeConnected(G, k);
> end for;
false
```

162.14 Distances, Paths, and Circuits in a Graph

The distance functions apply to graphs as well as to digraphs.

162.14.1 Distances, Paths, and Circuits in a Possibly Weighted Graph

There are a few distance and path functions that take into account the fact that the edges of the graph under consideration may be weighted. If the graph is not weighted, then the distance between two vertices is taken to be the length of the shortest path between the vertices. For more details on distance and paths functions in weighted graphs, refer to Section *Distances, Shortest Paths, and Minimum Weight Trees* in Chapter *Multigraphs*.

`Reachable(u, v): GrphVert, GrphVert -> BoolElt`

Return `true` if and only if vertices u and v of a graph G are in the same component of G.

`Distance(u, v): GrphVert, GrphVert -> RngIntElt`

Given two vertices u and v belonging to a graph G, return the length of a shortest path from u to v. If the graph is weighted, the function returns the total weight of the path from u to v. Results in an error if there is no path in G from u to v.

`Geodesic(u, v): GrphVert, GrphVert -> [GrphVert]`

Given vertices u and v belonging to the graph G, return a sequence of vertices $u = v_1, v_2, \ldots, v_n = v$ such that the sequence of edges $v_1v_2, v_2v_3, \ldots v_{n-1}v_n$ forms a shortest path joining u and v. Results in an error if there is no path in G from u to v.

162.14.2 Distances, Paths, and Circuits in a Non-Weighted Graph

The *distance* between two vertices is understood in its usual meaning of being the length of a shortest path between the vertices. The *diameter* of a graph is the maximum of the distances between every pair of vertices in the graph. Alternatively, it is the length of the longest geodesic in the graph. The distance and path functions listed below do not take account of the fact that the graph edges may be weighted.

`Diameter(G): Grph -> RngIntElt`

The maximum of the distances between every two vertices in the graph G is returned. If G is not connected, the value -1 is returned.

`DiameterPath(G): Grph -> [GrphVert]`

A sequence of vertices defining a longest geodesic in the graph G, if the G is connected, or the empty sequence if G is not connected.

`Ball(u, n): GrphVert, RngIntElt -> { GrphVert}`

Given a vertex u belonging to a graph G, and a non-negative integer n, return the set of vertices of G that are at distance n or less from u.

`Sphere(u, n): GrphVert, RngIntElt -> { GrphVert}`

Given a vertex u belonging to the graph G, and a non-negative integer n, return the set of vertices of G that are at distance n from u.

`DistancePartition(u): GrphVert -> [ { GrphVert} ]`

Given a vertex u belonging to the graph G, return the partition $P_0 \cup P_1 \cup \ldots \cup P_d$ of the vertex-set of G, where P_i is the set of vertices lying at distance i from vertex u. The partition is returned as a sequence of sets. Any vertices not connected to u are treated as being at infinite distance from u and are returned as the last set of the partition.

`IsEquitable(G, P): GrphUnd, { { GrphVert}} -> BoolElt`

`IsEquitable(G, P): GrphUnd, { { RngIntElt}} -> BoolElt`

Given a partition P of the vertex-set $V(G)$ of the graph G, return the value `true` if P is an equitable partition.

`EquitablePartition(P, G): { { GrphVert}}, GrphUnd -> { { GrphVert}}`

`EquitablePartition(P, G): { { RngIntElt}}, GrphUnd -> { { GrphVert}}`

Given a partition P of the vertex-set $V(G)$ of the graph G, return the coarsest partition of $V(G)$ that refines P and which is equitable.

`Girth(G): GrphUnd -> RngIntElt`

The girth of graph G, i.e. the length of a shortest cycle. To see how the automorphism group of G is computed see Subsection *Graph Colouring and Automorphism Group*.

`GirthCycle(G): GrphUnd -> [GrphVert]`

A cycle of shortest length in the graph G. To see how the automorphism group of G is computed see Subsection *Graph Colouring and Automorphism Group*.

162.15 Maximum Flow, Minimum Cut, and Shortest Paths

Whenever the edges of a graph are assigned a capacity it is possible to compute the maximum flow and the minimum cut of the graph. Whenever the edges of a graph are assigned a weight it is possible to find shortest paths in the graph. If one wants to perform these computations on a graph where none of its edges have been assigned a capacity or a weight, then the edges are taken to have a capacity or weight of one, and the computations are carried out accordingly.

The flow-based and shortest-paths algorithms and the resulting functionalities are fully documented in Chapter *Multigraphs*, Sections *Maximum Flow and Minimum Cut* and *Distances, Shortest Paths, and Minimum Weight Trees* respectively.

162.16 Matrices and Vector Spaces Associated with a Graph or Digraph

`AdjacencyMatrix(G): Grph -> AlgMatElt`

Returns the adjacency matrix for the (p, q) graph G as an element of the matrix ring $M_p(Z)$.

`DistanceMatrix(G): Grph -> AlgMatElt`

Returns the distance matrix A for the (p, q) graph G as an element of the matrix ring $M_p(Z)$. The (i, j)-th entry of A gives the distance between vertices v_i and v_j of G.

`IncidenceMatrix(G): Grph -> ModHomElt`

Returns the incidence matrix M for the (p, q) graph G as an element of the matrix bimodule $M^{p\times q}(Z)$.

If G is a graph, then entry (i, j) of M is 1 if the vertex v_i of G lies on the edge e_j of G. Otherwise entry (i, j) is zero.

If G is a digraph, entry (i, j) of M is 1 if vertex v_i is the initial vertex of the edge e_j, and -1 if v_i is the final vertex of the edge e_j. Otherwise entry (i, j) is zero. If e_j is a loop, then entry (i, j) may be either 1 or -1.

`IntersectionMatrix(G, P): GrphUnd, { { GrphVert}} -> AlgMatElt`

`IntersectionMatrix(G, P): GrphUnd, { { RngIntElt}} -> AlgMatElt`

Given an ordered equitable partition $P = P_1 \cup P_2 \cup \ldots \cup P_r$ of the vertex-set of the graph G, return the intersection matrix T for the partition. Thus, entry $T[i, j]$ is the number of vertices of the set P_j that are adjacent to a vertex of the set P_i.

162.17 Spanning Trees of a Graph or Digraph

`SpanningTree(G): GrphUnd -> Grph, GrphVertSet, GrphEdgeSet`

Given a connected (undirected) graph G, construct a spanning tree for G rooted at an arbitrary vertex of G. The spanning tree is returned as a subgraph of G. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

`SpanningForest(G): Grph -> Grph, GrphVertSet, GrphEdgeSet`

Given a graph G, construct a spanning forest for G. The forest is returned as a subgraph of G. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

`BreadthFirstSearchTree(u): GrphVert -> Grph, GrphVertSet, GrphEdgeSet`

`BFSTree(u): GrphVert -> Grph`

Given a vertex u belonging to the graph G, return a breadth-first search for G rooted at the vertex u. The tree is returned as a subgraph of G. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph. Note that G may be disconnected.

`DepthFirstSearchTree(u): GrphVert -> Grph, GrphVertSet, GrphEdgeSet, SeqEnum`

`DFSTree(u): GrphVert -> Grph, GrphVertSet, GrphEdgeSet, SeqEnum`

Given a vertex u belonging to the graph G, return a depth-first search tree T for G rooted at the vertex u. The tree T is returned as a subgraph of G. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph. Note that G may be disconnected.

The fourth return argument returns, for each vertex u of G, the tree order of u, that is, the order in which the vertex u has been visited while performing the depth-first search. If T does not span G then the vertices of G not in T are given tree order from `Order(T) + 1` to `Order(G)`.

162.18 Directed Trees

In the following functions, the graph is assumed to be a directed tree. This means it is a tree containing a root vertex and all edges are directed away from that vertex.

`IsRootedTree(G): GrphDir -> BoolElt, GrphVert`

Returns `true` exactly when the directed graph G is a tree having a vertex v such that all edges are directed away from v. In this case, the root vertex v is returned as a second value.

`Root(G): GrphDir -> GrphVert`

The root vertex of a rooted tree.

`IsRoot(v): GrphVert -> BoolElt`

Returns `true` if and only if the graph containing the vertex v is directed as a rooted tree with v as root.

`RootSide(v): GrphVert -> GrphVert`

When the graph containing the vertex v is directed as a rooted tree, this returns the unique neighbouring vertex to v which is closer to the root vertex. If v is the root vertex, it is returned itself.

`VertexPath(u, v): GrphVert, GrphVert -> SeqEnum`

A sequence of vertices comprising a path in a directed graph from the vertex u to the vertex v. The path does not necessarily respect the edge directions. Indeed it will first trace back to a common ancestor of u and v and then follow edge directions to v.

`BranchVertexPath(u, v): GrphVert, GrphVert -> SeqEnum`

The sequence of vertices on the vertex path from the vertex u to the vertex v having valency at least 3.

162.19 Colourings

The functions given here are not applicable to digraphs.

`ChromaticNumber(G): GrphUnd -> RngIntElt`

The chromatic number of the graph G, that is, the minimum number of colours required to colour the vertices of G such that no two adjacent vertices share the same colour.

`OptimalVertexColouring(G): GrphUnd -> SeqEnum`

An optimal vertex colouring of the graph G as a sequence of sets of vertices of G. Each set in the sequence contains the vertices which are given the same colour. Some optimal vertex colouring is returned, others may exist.

`ChromaticIndex(G): GrphUnd -> RngIntElt`

The chromatic index of the graph G, that is, the minimum number of colours required to colour the edges of G such that no two adjacent edges share the same colour.

`OptimalEdgeColouring(G): GrphUnd -> SeqEnum`

An optimal edge colouring of the graph G as a sequence of sets of edges of G. Each set in the sequence contains the edges which are given the same colour. Some optimal edge colouring is returned, others may exist.

`ChromaticPolynomial(G): GrphUnd -> RngUPolElt`

The chromatic polynomial of the graph G. For a given natural number x, the chromatic polynomial $p_G(x)$ gives the number of colourings of G with colours $1, 2, \ldots, x$.

Example: Chromatic Number (ex-8c2867)

We illustrate the use of these functions with the following graph:

```
> G:=Graph< 5 | [{2,5}, {1,3,5}, {2,4,5}, {3,5}, {1,2,3,4} ]>;
> ChromaticNumber(G);
3
> OptimalVertexColouring(G);
[
  { 1, 3 },
  { 2, 4 },
  { 5 }
]
> ChromaticIndex(G);
4
> OptimalEdgeColouring(G);
[
  { {1, 2}, {3, 5} },
  { {2, 3}, {1, 5} },
  { {3, 4}, {2, 5} },
  { {4, 5} }
]
> ChromaticPolynomial(G);
$.1^5 - 7*$.1^4 + 18*$.1^3 - 20*$.1^2 + 8*$.1
```

162.20 Cliques, Independent Sets

The functions given here are not applicable to digraphs.

A *clique* of a graph G is a complete subgraph of G. A *maximal clique* is a clique which is not contained in any other clique. A *maximum clique* is a maximal clique of largest size.

An *independent set* of G is an empty subgraph of G. A *maximal* (*maximum*) *independent set* is defined is the same way as a maximal (maximum) clique. Note that an independent set of size k in the graph G is a clique of size k in the complement graph of G.

The clique and independent set functions presented below are implemented using one of two algorithms, called here *BranchAndBound* and *Dynamic*. The algorithms *BranchAndBound* and *Dynamic* are briefly described here.

BranchAndBound : The algorithm is an implementation of the branch and bound algorithm by Bron and Kerbosch [BK73].

The algorithm is designed to find maximal cliques and it has been adapted here to also find cliques of specific size which may not be maximal. It attempts to build a maximal clique by trying to expend the current maximal clique. Some heuristics are built into the algorithm which enable to prune the search tree.

Dynamic : The algorithm finds a clique of size exactly k, not necessarily maximal, in the graph G by using recursion and dynamic programming. If a clique of size k exists in G, then, for a given vertex v of G, there is either a clique of size $k-1$ in the subgraph induced by the neighbours of v, or there is a clique of size k in the graph $G-v$. This is the recursive step. The *Dynamic* algorithm applies a different strategy (sorting of vertices and selection of vertices to consider) according to the order and density of the subgraph considered at the current level of recursion. This is achieved by dynamic programming, hence the name. This algorithm is due to Wendy Myrvold [MPW].

Comment: When comparing both algorithms in the situation where the problem is to find a maximum clique one observes that in general *BranchAndBound* does better. However *Dynamic* outperforms *BranchAndBound* when the graphs under consideration are large (more then 400 vertices) random graphs with high density (larger than 0.5%). So far, it can only be said that the comparative behaviour of both algorithms is highly dependent on the structure of the graphs.

`HasClique(G, k): GrphUnd, RngIntElt -> BoolElt, { GrphVert }`

Returns `true` if and only if the graph G has a maximal clique of size k. If `true`, returns such a clique as the second argument.

`HasClique(G, k, m : parameters): GrphUnd, RngIntElt, BoolElt -> BoolElt, { GrphVert }`

`Al: MonStgElt`	`Default: "BranchAndBound"`

If m is `true`, the function is `true` if and only if the graph G has a maximal clique of size k. If m is `false`, the function is `true` if and only if the graph G has a — not necessarily maximal — clique of size k. If the function is `true`, an appropriate clique is returned as the second argument. When m is `false`, the parameter `Al` enables the user to select the algorithm which is to be used. When m is `true`, the parameter `Al` is ignored.

The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "BranchAndBound"` or `Al := "Dynamic"`. See the description given at the beginning of this section. The default is `"BranchAndBound"`.

`HasClique(G, k, m, f : parameters): GrphUnd, RngIntElt, BoolElt, RngIntElt -> BoolElt, { GrphVert }`

`Al: MonStgElt`	`Default: "BranchAndBound"`

If m is `true` and $f = 0$, the function is `true` if and only if the graph G has a maximal clique of size k. If m is `true` and $f = 1$, the function is `true` if and only if the graph G has a maximal clique of size larger than or equal to k. If m is `true` and $f = -1$, the function is `true` if and only if the graph G has a maximal clique of size smaller than or equal to k. If m is `false`, the function is `true` if and only if the graph G has a — not necessarily maximal — clique of size k. If the function is `true`, an appropriate clique is returned as the second argument. When m is `false`, the parameter `Al` enables the user to select the algorithm which is to be used. When m is `true` the parameter `Al` is ignored, and when m is `false`, the flag f is ignored.

The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "BranchAndBound"` or `Al := "Dynamic"`. See the description given at the beginning of this section. The default is `"BranchAndBound"`.

`MaximumClique(G : parameters): GrphUnd -> { GrphVert }`

`Al: MonStgElt`	`Default: "BranchAndBound"`

Finds a maximum clique in the graph G. The clique is returned as a set of vertices.

The parameter `Al` enables the user to select the algorithm which is to be used. For a description of the algorithm used when `Al := "BranchAndBound"` see the beginning of this section. When `Al := "Dynamic"`, two steps are required.

Step 1
Finding a lowerbound on the size of a maximum clique. This is achieved by using the *dsatur* colouring (*dsatur* stands for saturation degree) due to Brélaz [Bre79]. The *dsatur* colouring gives a reasonably good approximation to the size of a maximum clique, usually with a penalty of 2 to 3.

Step 2
Assume that the lowerbound found in Step 1 if l. Then a maximum clique is found by finding the largest possible clique of size $k \geq l$ using the Dynamic algorithm.

The default is `"BranchAndBound"`.

`CliqueNumber(G : parameters): GrphUnd -> RngIntElt`

```
Al: MonStgElt                    Default: "BranchAndBound"
```

Finds the size of a maximum clique in the graph G. The parameter `Al` enables the user to select the algorithm which is to be used. For a description of the algorithm used when `Al := "BranchAndBound"` see the beginning of this section. For a description of the algorithm used when `Al := "Dynamic"` see the function `MaximumClique`. The default is `"BranchAndBound"`.

`AllCliques(G : parameters): GrphUnd -> SeqEnum`

```
Limit: RngIntElt                 Default: 0
```

Returns all maximal cliques of the graph G as a sequence of sets of vertices. If `Limit` is set to a positive integer, returns `Limit` maximal cliques of G.

`AllCliques(G, k : parameters): GrphUnd, RngIntElt -> SeqEnum`

```
Limit: RngIntElt                 Default: 0
```

Returns all maximal cliques of size k of the graph G as a sequence of sets of vertices. If `Limit` is set to a positive integer, returns `Limit` maximal cliques of size k of G.

`AllCliques(G, k, m : parameters): GrphUnd, RngIntElt, BoolElt -> SeqEnum`

```
Limit: RngIntElt                 Default: 0
Al   : MonStgElt                 Default: "BranchAndBound"
```

If m is `true`, returns all maximal cliques of size k in the graph G. If m is `false`, returns all — not necessarily maximal — cliques of size k. When m is `false`, the parameter `Al` enables the user to select the algorithm which is to be used. When m is `true`, the parameter `Al` is ignored.

The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "BranchAndBound"` or `Al := "Dynamic"`. See the description given at the beginning of this section. The default is `"BranchAndBound"`.

Except in the case where m is `false` and `Al` is `"Dynamic"`, the parameter `Limit`, if set to a positive integer, limits the number of cliques returned.

Maximal independent sets or independent sets of a given size k in a graph G can be easily found by finding maximal cliques or cliques of size k in the complement of G. Only two functions which are concerned with independent sets are provided: one finds a maximum independent set and the other returns the independence number of a graph.

`MaximumIndependentSet(G: parameters): GrphUnd -> { GrphVert }`

```
Al: MonStgElt                    Default: "BranchAndBound"
```

Finds a maximum independent set in the graph G. The maximum independent set is returned as a set of vertices.

The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "BranchAndBound"` or `Al := "Dynamic"`. See the function `MaximumClique`. The default is `"BranchAndBound"`.

`IndependenceNumber(G: parameters): GrphUnd -> RngIntElt`

```
Al: MonStgElt                    Default: "BranchAndBound"
```

Finds the size of a maximum independent set in the graph G.

The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "BranchAndBound"` or `Al := "Dynamic"`. See the function `CliqueNumber`. The default is `"BranchAndBound"`.

Example: Cliques of a Graph (ex-8cd121)

We illustrate the use of the clique functions with the following graph:

```
> G := Graph< 9 | [ {4,5,6,7,8,9}, {4,5,6,7,8,9}, {4,5,6,7,8,9},
>                   {1,2,3,7,8,9}, {1,2,3,7,8,9}, {1,2,3,7,8,9},
>                   {1,2,3,4,5,6}, {1,2,3,4,5,6}, {1,2,3,4,5,6} ]>;
> HasClique(G, 2);
false
> HasClique(G, 2, true);
false
> HasClique(G, 2, false);
true { 1, 4 }
> HasClique(G, 2, true: Al := "Dynamic");
false
> HasClique(G, 2, false: Al := "Dynamic");
true { 1, 9 }
> HasClique(G, 2, true, 1);
true { 1, 4, 7 }
> MaximumClique(G);
{ 1, 4, 7 }
> AC := AllCliques(G);
> #AC;
27
> AC3 := AllCliques(G,3);
> #AC3;
27
> AC eq AC3;
true
> AllCliques(G, 2, true);
[]
> AllCliques(G, 2, false);
[
  { 1, 4 },
  { 1, 5 },
  { 1, 6 },
  { 1, 7 },
  { 1, 8 },
  { 1, 9 },
  { 2, 4 },
```

(continues on next page)

(continued from previous page)

```
  { 2, 5 },
  { 2, 6 },
  { 2, 7 },
  { 2, 8 },
  { 2, 9 },
  { 3, 4 },
  { 3, 5 },
  { 3, 6 },
  { 3, 7 },
  { 3, 8 },
  { 3, 9 },
  { 4, 7 },
  { 4, 8 },
  { 4, 9 },
  { 5, 7 },
  { 5, 8 },
  { 5, 9 },
  { 6, 7 },
  { 6, 8 },
  { 6, 9 }
]
```

162.21 Planar Graphs

A linear-time algorithm due to Boyer and Myrvold [BM01] has been implemented to test if a graph is planar, and to identify a Kuratowski subgraph if the graph is non-planar. This algorithm requires that the graph has a sparse representation.

The interest of this new algorithm is that it is very easy to implement when compared to previous linear-time planarity testers. In addition, the algorithm also isolates a Kuratowski subgraph, if any, in linear-time.

The concept underlying the tester is deceptively simple: the idea is to perform a depth first search of the graph and then to try to embed (by post-order traversal of the tree) the back edges from a vertex v to its descendants in such a way that they do not cross and such that vertices with connections to ancestors of v remain along the external face.

To decide the order in which to embed back edges from v to its descendants, the external faces of components rooted by children of v are explored top-down, avoiding paths containing vertices with ancestor connections. To decide whether a vertex u has ancestor connections and must therefore be kept on the external face, one only needs the least ancestor directly adjacent to u and the lowpoint information for the children of u that are separable from the parent of u in the partial embedding.

Each back edge $[w, v]$ is added as soon as the descendant endpoint w is encountered. If w is not in the same biconnected component as v, then the biconnected components encountered during the traversal are merged at their respective cut vertices since the new edge $[w, v]$ biconnects them. The merge is performed such that vertices with ancestor connections remain on the external face of the newly formed biconnected component.

Traversal is terminated when all back edges from v to its descendants are added or when the external face traversal is prevented from reaching the descendant endpoint of a back edge. This occurs when both external face paths extending from the least numbered vertex in a biconnected component contain vertices with ancestor connections that block further traversal to a vertex that is the descendant endpoint w of a back edge $[w, v]$. In this case, a Kuratowski subgraph is isolated based primarily on ancestor connections, external face paths, and depth first search tree paths.

Magma is deeply grateful for the valuable help and collaboration from the authors of [BM01], in particular John Boyer.

`IsPlanar(G): GrphUnd -> BoolElt, GrphUnd`

Tests whether the (undirected) graph G is planar. The graph may be disconnected. If the graph is non-planar then a Kuratowski subgraph of G is returned: That is, a subgraph of G homeomorphic to K_5 or $K_{3,3}$. The support and vertex/edge decorations of G are *not* retained in this (structural) subgraph.

`Obstruction(G): GrphUnd -> GrphUnd`

Returns a Kuratoswki obstruction if the graph is non-planar, or the empty graph if the graph is planar. The Kuratoswki is a (structural) subgraph of G; the support and vertex/edge decorations of G are not transferred to it.

`IsHomeomorphic(G : parameters): GrphUnd -> BoolElt`

Graph: MonStgElt	Default:

Tests if a graph is homeomorphic to either K_5 or $K_{3,3}$. The parameter `Graph` must be set to either "K5" or "K33"; it has no default setting.

`Faces(G): GrphUnd -> SeqEnum[GrphVert]`

Returns the faces of the planar graph G as sequences of the edges bordering the faces of G. If G is disconnected, then the face defined by an isolated vertex v is given as $[v]$.

`Face(u, v): GrphVert, GrphVert -> SeqEnum`

Returns the face of the planar graph G bordered by the directed edge $[u, v]$ as an ordered list of edges of G.

Note that a directed edge and an orientation determine a face uniquely: We can assume without loss of generality that the plane is given a clockwise orientation. Then given a directed edge $e = [u_1, v_1]$, the face defined by e is the ordered set of edges $[u_1, v_1], [u_2, v_2], \ldots, [u_m, v_m]$ such that $v_i = u_{i+1}$ for all i, $1 \leq i < m$, $v_m = u_1$, and for each $v_i = u_{i+1}$, the neighbours of v_i, u_i and v_{i+1}, are *consecutive* vertices in v_i's adjacency list whose order is *anti-clockwise*.

`Face(e): GrphEdge -> SeqEnum`

Let e be the edge u, v of the planar graph G (recall that G is undirected). Then `Face(`u, v`)` returns the face bordered by the directed edge $[u, v]$ as a sequence of edges of G.

`NFaces(G): GrphUnd -> RngIntElt`

`NumberOfFaces(G): GrphUnd -> RngIntElt`

Returns the number of faces of the planar graph G. In the case of a disconnected graph, an isolated vertex counts for one face.

`Embedding(G): GrphUnd -> SeqEnum`

Returns the graph G's planar embedding as a sequence S where $S[i]$ is a sequence of edges incident from vertex i.

`Embedding(v): GrphVert -> SeqEnum`

Returns the ordered list of edges (in clockwise order say) incident from vertex v.

`PlanarDual(G): GrphUnd -> GrphUnd`

Constructs the dual G' of a planar graph G. The numbering of the vertices of G' corresponds to the order of the faces as returned by `Faces(G)`.

Example: Faces of a Planar Graph (ex-df235a)

We start with a small disconnected planar graph G: Notice how one of the faces of G is defined by the isolated vertex 4.

```
> G := Graph< 5 | [ {2, 3, 5}, {1, 5}, {1}, {}, {1, 2} ] >;
> G;
Graph
Vertex  Neighbours
```

(continues on next page)

(continued from previous page)

```
1       2 3 5 ;
2       1 5 ;
3       1 ;
4       ;
5       1 2 ;

> IsConnected(G);
false
> IsPlanar(G);
true
> Faces(G);
[
  [ {1, 2}, {2, 5}, {5, 1} ],
  [ {1, 5}, {5, 2}, {2, 1}, {1, 3}, {3, 1} ],
  [ 4 ]
]
> Embedding(G);
[
  [ {1, 2}, {1, 5}, {1, 3} ],
  [ {2, 5}, {2, 1} ],
  [ {3, 1} ],
  [],
  [ {5, 1}, {5, 2} ]
]
> Embedding(VertexSet(G)!1);
[ {1, 2}, {1, 5}, {1, 3} ]
```

Now let's turn to a simple non-planar graph whose obstruction is $K_{3,3}$:

```
> G := Graph< 6 | [ {3, 4, 5}, {3, 4, 5, 6}, {1, 2, 5, 6},
> {1, 2, 5, 6}, {1, 2, 3, 4, 6}, {2, 3, 4, 5} ] >;
> G;
Graph
Vertex  Neighbours

1       3 4 5 ;
2       3 4 5 6 ;
3       1 2 5 6 ;
4       1 2 5 6 ;
5       1 2 3 4 6 ;
6       2 3 4 5 ;

> IsPlanar(G);
false
> b, O := IsPlanar(G);
> IsSubgraph(G, O);
true
> O;
Graph
Vertex  Neighbours

1       4 5 3 ;
2       3 5 4 ;
3       1 6 2 ;
4       2 6 1 ;
```

(continues on next page)

(continued from previous page)

```
5       6 1 2 ;
6       5 4 3 ;

> IsHomeomorphic(O : Graph := "K33");
true
```

162.22 Automorphism Group of a Graph or Digraph

The automorphism group functionality is an interface to version 2.6r7 of the `nauty` and `Traces` packages of B. McKay and A. Piperno. For a paper describing `nauty` see [McK81]. There also exists a user's manual [MP] describing the essential features of `nauty` and `Traces`.

162.22.1 The Automorphism Group Function

`AutomorphismGroup(G : parameters): Grph -> GrpPerm, GSet, GSet, PowMap, Map, Grph`

```
Al          : MonStgElt                          Default: "nauty"
Canonical   : BoolElt                            Default: false
Stabilizer  : [ { GrphVert } ]                   Default:
Invariant   : MonStgElt                          Default:
Minlevel    : RngIntElt                          Default: 0
Maxlevel    : RngIntElt                          Default:
Arg         : RngIntElt                          Default:
Print       : RngIntElt                          Default: 0
IgnoreLabels: BoolElt                            Default: false
```

Compute the automorphism group of the graph G. This returns the group A, the vertex and edge G-sets (in that order), the power structure $\mathcal{A}$ of all automorphisms of G, and the transfer map from A to $\mathcal{A}$. Note that G may be directed or undirected. For small graphs (i.e. having less than 500 vertices) the canonically labelled graph is also returned.

The automorphism group computation may be driven by several user parameters. There are five principal parameters: `Al`, `Canonical`, `Stabilizer`, `Invariant`, and `Print`. Some of these parameters have associated parameters.

The parameter `Al` controls whether the `nauty` or `Traces` algorithms are used to compute the automorphism group. Some parameters are only applicable with the `nauty` algorithm.

If the parameter `Canonical` is set to `true`, then the canonically labelled graph will be returned. If the graph has fewer than 500 vertices then the default value for this parameter is `true`, while if the graph has 500 or more vertices, the default value is `false`. It is important to note that the canonically labelled graph is *dependent* upon all the parameters used to compute the automorphism group, including the algorithm parameter.

If the parameter `Stabilizer` is assigned a partition P of the vertex-set of G as its value, then the subgroup of the automorphism group of G that preserves the partition P will be computed. The partition given as `Stabilizer` need not be a partition of the full vertex-set of G. Any vertices not covered by the sets in `Stabilizer` are assumed to belong to an extra partition cell.

This parameter is only relevant when the `nauty` algorithm is used. It causes the algorithm to use the named invariant to assist the auto group computation. The invariant is specified by a string which is the name of a C function computing an invariant, as on pages 37–39 of the `nauty` and `Traces` manual [MP]. The default value for `Invariant` is `"Null"` when G is an undirected graph, or `"adjacencies"` when G is a digraph. For a more detailed discussion on invariants see Subsection *nauty Invariants*.

There are three optional associated parameters to `Invariant`:

An expression, depending upon the type of invariant. When G is undirected these three parameters take the values 0, 1, and 0 respectively. When G is a digraph they take the values 0, 2, and 0 respectively. When `Invariant` is one of "indsets", "cliques", "cellcliq", or "cellind", the default value for `Arg` is 3. Again, see Subsection *nauty Invariants* for more information.

The integer valued parameter `Print` is used to control informative printing according to the following table:

`Print := 0` : No output (default).

`Print := 1` : Statistics are printed.

`Print := 2` : Summary upon completion of each level (`nauty` only).

`Print := 3` : Print the automorphisms as they are discovered.

By default, the automorphism group computation respects vertex labels (colours). That is, elements of the group will only take a vertex to an identically labelled vertex. The Boolean value `IgnoreLabels` will treat all vertices as identically labelled for the purposes of this computation.

162.22.2 nauty Invariants

Invariants can be used to supplement the built-in partition refinement code in `nauty`, to assist in the processing of difficult graphs. Invariants are used with three parameters:

`Minlevel`: the minimum depth in the search tree at which an invariant is to be applied (default 0)

`Maxlevel`: the maximum depth in the search tree at which an invariant is to be applied (default 1 for undirected graphs and 2 for digraphs)

`Arg`: an integer argument which has a different meaning (or no meaning) for each invariant

The root of the tree, corresponding to the partition provided by the user, has level 1. Note that the invariants use an existing partition and attempt to refine it. Thus it can be that an invariant fails to refine the partition at the root of the search tree but succeeds further down the tree. In particular, all invariants necessarily fail at the root of the tree if the graph is vertex-transitive.

The invariants are :

`"default"`

:

- Default — No invariant is used.

`"twopaths"`

:

- Each vertex is classified according to the vertices reachable from along a path of length 2. This is a cheap invariant sometimes useful for regular graphs. `Arg` is not used.

`"adjtriang"`

:

- This counts the number of common neighbours between pairs of adjacent (if `Arg` $= 0$) or non–adjacent (if `Arg` $= 1$) vertices. This is a fairly cheap invariant which can often work for strongly regular graphs.

`"triples"`

:

- This considers in detail the neighbourhoods of each triple of vertices. It often works for strongly regular graphs that `"adjtriang"` fails on, but is more expensive. `Arg` is not used.

`"quadruples"`

:

- Similar to triples but using quadruples of vertices. Much more expensive but can work on graphs with highly regular structure that `"triples"` fails on. `Arg` is not used.

`"celltrips"`

:

- Like `"triples"`, but only triples inside one cell are used. One to try for the bipartite incidence graphs of structures like block designs (but won't work for projective planes). `Arg` is not used.

`"cellquads"`

:

- Like `"quadruples"`, but only quadruples inside one cell are used. Originally designed for some exceptional graphs derived from Hadamard matrices (where applying it at level 2 is best). `Arg` is not used.

`"cellquins"`

:

- Considers 5-tuples of vertices inside a cell. Very expensive. We don't know of a good application. `Arg` is not used.

`"cellfano"`

:

`"cellfano2"`

:

- These are intended for the bipartite graphs of projective planes and similar highly regular structures. `"cellfano2"` is cheaper, so try it first. `Arg` is not used. The method is similar to counting Fano subplanes.

`"distances"`

:

- This uses the distance matrix of the graph. It is good at partitioning general regular graphs (but not strongly regular graphs). Moderately cheap. `Arg` is unused.

`"indsets"`

:

- This uses the independent sets of size `Arg` (default 3). Can be successful for strongly regular graphs (for `Arg` ≥ 4) or regular bipartite graphs.

`"cliques"`

:

- This uses the cliques of size `Arg` (default 3). Can be successful for strongly regular graphs (for `Arg` ≥ 4).

`"cellind"`

:

- This uses the independent sets of size `Arg` (default 3) that lie entirely within one cell of the partition. Try applying at level 2 for difficult vertex-transitive graphs.

`"cellcliq"`

:

- This uses the cliques of size `Arg` (default 3) that lie entirely within one cell of the partition. Try applying at level 2 for difficult vertex-transitive graphs.

"adjacencies"

:

- This is a simple invariant that corrects for a deficiency in the built-in partition refinement procedure for directed graphs. It is the default in Magma for directed graphs. Apply at a few levels, say levels 1–5. Arg is unused.

These invariants are further described on pages 37–39 of the nauty and Traces manual [MP]. Also, we provide a facility enabling users to test if a specific Invariant achieves a partition refinement for a graph G.

IsPartitionRefined(G: parameters): Grph -> BoolElt

```
Stabilizer  : [ { Vert } ]                  Default:
Invariant   : MonStgElt                     Default:
Arg         : RngIntElt                     Default:
IgnoreLabels: BoolElt                       Default: false
```

Returns true if and only the invariant in Invariant could refine the graph G's vertex-set partition. If no invariant is set in Invariant then the refinement procedure which is tested is taken to be the default one.

Four parameters can be passed:

The parameters have the same meaning as for *AutomorphismGroup*.

162.22.3 Graph Colouring and Automorphism Group

For any calls to nauty or Traces via Magma functions the graph colouring of a graph G (as set by *AssignLabels* or a similar function) is taken as an *intrinsic* property of G. Consequently, the default automorphism group computed by nauty or Traces is the group for the coloured graph G. In particular this implies that, when G is coloured, this default automorphism group is *not* G's full automorphism group.

There are several Magma functions which make use of the automorphism group of a graph, and therefore rely on a call to nauty. We list them here:

1 Diameter, DiameterPath, IsDistanceRegular, GirthCycle, Girth

2 IsDistanceTransitive, IsTransitive, IsPrimitive, IsSymmetric, IsIsomorphic, IsEdgeTransitive, EdgeGroup, OrbitsPartition, CanonicalGraph

The functions in Group 1 make use of the automorphism group of the graph only as a means to improve efficiency. If some group has already been computed for the graph under consideration then this group will be used. Otherwise the graph's default group is computed.

The functions in Group 2 all use the graph's default group. If a group A of the graph G has already been computed but is not G's default group (for example if a stabilizer of G was given when computing A) then the graph's default group will be computed. Consequently, as an example, if G is coloured, IsTransitive(G) will (always) return false.

For all these functions it is possible to drive the group computation by setting an appropriate Invariant parameter. This is particularly useful when it is known that some invariant will speed up the group computation. This is achieved by first computing the group using *AutomorphismGroup* while setting Invariant. This parameter is then remembered for any further (re)computation of the default automorphism group, unless reset by a subsequent call to AutomorphismGroup. For convenience, the function IsIsomorphic accepts a parameter suite very similar to that of the AutomorphismGroup function: It is particularly useful when stabilizers need to be set.

162.22.4 Variants of Automorphism Group

`CanonicalGraph(G): Grph -> Grph`

Given a graph G, return the canonically labelled graph isomorphic to G. To see which automorphism group is computed see Subsection *Graph Colouring and Automorphism Group*. Users should be aware that the canonical graph is *dependent* upon the invariant used when computing the automorphism group. Thus, the computation of the automorphism group of G using different invariants will produce *different* canonical labellings.

`EdgeGroup(G): Grph -> GrpPerm, GSet`

Returns the automorphism group of the graph G in its action on the edges of G, and the G-set of edges of G. To see which automorphism group is computed see Subsection *Graph Colouring and Automorphism Group*.

`IsIsomorphic(G, H : parameters): GrphDir, GrphDir -> BoolElt, Map`

`IsIsomorphic(G, H : parameters): GrphUnd, GrphUnd -> BoolElt, Map`

```
Stabilizer  : [ { Vert } ]                  Default:
Stabilizer2 : [ { Vert } ]                  Default:
IgnoreLabels: BoolElt                       Default: false
```

This function returns `true` if the graphs G and H are isomorphic. If G and H are isomorphic, a second value is returned, namely a mapping between the vertices of G and the vertices of H giving the isomorphism.

The parameters accepted by `IsIsomorphic` are almost the same as for *AutomorphismGroup*. The exceptions are:

This sets the stabilizer set for *both* graphs G and H. If one wishes to set two distinct stabilizers then one would also use:

This sets the stabilizer set for the second graph H only.

This indicates whether or not to ignore the graph labelling for *both* graphs G and H.

Graph isomorphism is understood as mapping colours to colours and stabilizers to stabilizers. Consequently comparing two graphs for isomorphism will always return `false` if one graph is coloured while the other one is not. Similarly two graphs can't be isomorphic if each is coloured using a different set of colours. Moreover, it is a `nauty` requirement that in order to achieve mapping of stabilizers they must be compatible: that is, they *must* have same-sized cells in the same order.

In general, to see which and how the graphs' automorphism groups are computed see Subsection *Graph Colouring and Automorphism Group*.

Example: Automorphism Group and Canonical Graph `(ex-bda3f8)`

We provide a very simple example of the use of `AutomorphismGroup` which shows how the automorphism group and the corresponding canonical graph differ depending on the labelling of the graph and of the stabilizer given, if any.

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} }>;
>
> AssignLabels(VertexSet(G), ["a", "b", "a", "b", "b"]);
> A, _, _, _, _, C1 := AutomorphismGroup(G);
> A;
Permutation group A acting on a set of cardinality 5
Order = 2
  (1, 3)(4, 5)
> C1;
Graph
Vertex  Neighbours
```

(continues on next page)

(continued from previous page)

```
1       3 5 ;
2       4 5 ;
3       1 4 ;
4       2 3 ;
5       1 2 ;

> B, _, _, _, _, C2 := AutomorphismGroup(G : IgnoreLabels := true);
> B;
Permutation group B acting on a set of cardinality 5
Order = 10 = 2 * 5
  (2, 5)(3, 4)
  (1, 2)(3, 5)
> C2;
Graph
Vertex  Neighbours

1       2 3 ;
2       1 4 ;
3       1 5 ;
4       2 5 ;
5       3 4 ;

> C, _, _, _, _, C3 := AutomorphismGroup(G : Stabilizer :=
> [ { VertexSet(G) | 1, 2 } ]);
> C;
Permutation group C acting on a set of cardinality 5
> #C;
1
> C3;
Graph
Vertex  Neighbours

1       3 4 ;
2       3 5 ;
3       1 2 ;
4       1 5 ;
5       2 4 ;
```

We now check that the graph returned by `CanonicalGraph` is identical to C_1:

```
> C4 := CanonicalGraph(G);
> C4 eq C1;
true
```

Example: Isomorphism of Coloured Graphs (ex-4bd4fd)

These two examples should help clarify how `IsIsomorphic` behaves for graphs for which a colouring is defined. The first example shows that two graphs cannot be isomorphic if one is coloured while the other is not. This example also demonstrates the use of stabilizers.

```
> G1 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["a", "a", "b", "b", "b"]);
> G2 := CompleteGraph(5);
> IsIsomorphic(G1, G2);
false
> IsIsomorphic(G1, G2 : IgnoreLabels := true);
```

(continues on next page)

(continued from previous page)

```
true
>
> V1 := Vertices(G1);
> V2 := Vertices(G2);
> S1 := { V1 | 1, 2, 3};
> S2 := { V2 | 3, 4, 5};
>
> IsIsomorphic(G1, G2 : Stabilizer := [S1],
>       Stabilizer2 := [S2]);
false
> IsIsomorphic(G1, G2 : Stabilizer := [S1],
>       Stabilizer2 := [S2], IgnoreLabels := true);
true
> IsIsomorphic(G1, G2 : Stabilizer := [S1],
> IgnoreLabels := true);
true
>
> SS1 := [ { V1 | 1}, {V1 | 2, 3} ];
> SS2 := [ { V2 | 3, 4}, { V2 | 1} ];
> IsIsomorphic(G1, G2 : Stabilizer := SS1, Stabilizer2 := SS2,
> IgnoreLabels := true);
false
>
> SS1 := [ {V1 | 2, 3}, { V1 | 1} ];
> IsIsomorphic(G1, G2 : Stabilizer := SS1, Stabilizer2 := SS2,
> IgnoreLabels := true);
true
```

The second example shows that two graphs are isomorphic if and only if colours map to the *same* colours.

```
> G1 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["b", "b", "b", "a", "a"]);
> G2 := CompleteGraph(5);
> AssignLabels(VertexSet(G2), ["a", "a", "b", "b", "b"]);
> IsIsomorphic(G1, G2);
true
>
> G1 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["a", "b", "b", "c", "c"]);
> G2 := CompleteGraph(5);
> AssignLabels(VertexSet(G2), ["a", "c", "b", "b", "c"]);
> IsIsomorphic(G1, G2);
true
>
> G1 := CompleteGraph(5);
> G2 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["a", "a", "b", "b", "b"]);
> AssignLabels(VertexSet(G2), ["b", "b", "a", "a", "a"]);
> IsIsomorphic(G1, G2);
false
> IsIsomorphic(G1, G2 : IgnoreLabels := true);
true
>
> G1 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["b", "b", "b", "a", "a"]);
```

(continues on next page)

(continued from previous page)

```
> G2 := CompleteGraph(5);
> AssignLabels(VertexSet(G2), ["b", "b", "c", "c", "c"]);
> IsIsomorphic(G1, G2);
false
> IsIsomorphic(G1, G2 : IgnoreLabels := true);
true
```

162.22.5 Action of Automorphisms

The automorphism group A of a graph G is given in its action on the standard support and it does not act directly on G. The action of A on G is obtained using the G-set mechanism. The two basic G-sets associated with the graph correspond to the action of A on the set of vertices V and the set of edges V of G. These two G-sets are given as return values of the function `AutomorphismGroup` or may be constructed directly. Additional G-sets associated with a graph may be built using the G-set constructors. Given a G-set Y for A, the action of A on Y may be studied using the permutation group functions that allow a G-set as an argument In this section, only a few of the available functions are described: see the section on G-sets for a complete list.

`Image(a, Y, y): GrpPermElt, GSet, Elt -> Elt`

Let a be an element of the automorphism group A for the graph G and let Y be a G-set for A. Given an element y belonging either to Y or to a G-set derived from Y, find the image of y under a.

`Orbit(A, Y, y): GrpPerm, GSet, Elt -> GSet`

Let A be a subgroup of the automorphism group for the graph G and let Y be a G-set for A. Given an element y belonging either to Y or to a G-set derived from Y, construct the orbit of y under A.

`Orbits(A, Y): GrpPerm, GSet -> [ GSet ]`

Let A be a subgroup of the automorphism group for the graph G and let Y be a G-set for G. This function constructs the orbits of the action of A on Y.

`Stabilizer(A, Y, y): GrpPerm, GSet, Elt -> GrpPerm`

Let A be a subgroup of the automorphism group for the graph G and let Y be a G-set for A. Given an element y belonging either to Y or to a G-set derived from Y, construct the stabilizer of y in A.

`Action(A, Y): GrpPerm, GSet -> Hom(Grp), GrpPerm, GrpPerm`

Given a subgroup A of the automorphism group of the graph G, and a G-set Y for A, construct the homomorphism $\phi : A \to L$, where the permutation group L gives the action of A on the set Y. The function returns:

(a) The natural homomorphism $\phi : A \to L$;

(b) The induced group L;

(c) The kernel of the action (a subgroup of A).

`ActionImage(A, Y): GrpPerm, GSet -> GrpPerm`

Given a subgroup A of the automorphism group of the graph structure G, and a G-set Y for A, construct the permutation group L giving the action of A on the set Y.

`ActionKernel(A, Y): GrpPerm, GSet -> GrpPerm`

Given a subgroup A of the automorphism group of the graph G, and a G-set Y for A, construct the kernel of the action of A on the set Y.

Example: Automorphisms of the Clebsch Graph `(ex-ec7c11)`

We construct the Clebsch graph (a strongly regular graph) and investigate the action of its automorphism group.

```
> S := { 1 .. 5 };
> V := &join[ Subsets({1..5}, 2*k) : k in [0..#S div 2]];
> E := { {u,v} : u,v in V | #(u sdiff v) eq 4 };
> G, V, E := StandardGraph( Graph< V | E >);
> G;
Graph
Vertex  Neighbours

1       4 6 8 10 12 ;
2       5 6 9 12 14 ;
3       9 10 12 13 16 ;
4       1 7 9 14 16 ;
5       2 7 8 10 16 ;
6       1 2 13 15 16 ;
7       4 5 12 13 15 ;
8       1 5 9 11 13 ;
9       2 3 4 8 15 ;
10      1 3 5 14 15 ;
11      8 12 14 15 16 ;
12      1 2 3 7 11 ;
13      3 6 7 8 14 ;
14      2 4 10 11 13 ;
15      6 7 9 10 11 ;
16      3 4 5 6 11 ;
> A, AV, AE := AutomorphismGroup(G);
> A;
Permutation group aut acting on a set of cardinality 16
Order = 1920 = 2^7 * 3 * 5
    (2, 15)(5, 11)(7, 14)(10, 12)
    (3, 11)(8, 10)(9, 14)(13, 15)
    (2, 11)(5, 15)(6, 8)(9, 16)
    (2, 7)(4, 6)(9, 13)(14, 15)
    (1, 2, 13, 15)(3, 11, 4, 5)(7, 10, 12, 14)(8, 9)
> CompositionFactors(A);
    G
    |  Cyclic(2)
    *
    |  Alternating(5)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
> IsPrimitive(A);
true
```

From the composition factors we guess that the group is $Sym(5)$ extended by the elementary abelian group of order 16. Let us verify this and relate it to the graph. We begin by trying to get at the group of order 16. We ask for the Fitting subgroup.

```
> F := FittingSubgroup(A);
> F;
Permutation group F acting on a set of cardinality 16
```

(continues on next page)

(continued from previous page)

```
Order = 16 = 2^4
    (1, 13)(2, 11)(3, 4)(5, 15)(6, 8)(7, 10)(9, 16)(12, 14)
    (1, 10)(2, 9)(3, 12)(4, 14)(5, 8)(6, 15)(7, 13)(11, 16)
    (1, 11)(2, 13)(3, 5)(4, 15)(6, 14)(7, 9)(8, 12)(10, 16)
    (1, 12)(2, 6)(3, 10)(4, 7)(5, 16)(8, 11)(9, 15)(13, 14)
```

Since A is primitive, this looks as if it is an elementary abelian regular normal subgroup:

```
> EARNS(A) eq F;
true
```

Thus, A has a regular normal subgroup N of order 16 which acts transitively on the vertices. Therefore, the stabilizer of any point is a complement of N.

```
> S1 := Stabilizer(A, AV, V!1);
> #S1;
120
> IsTransitive(S1);
false
> Orbits(S1);
[
    GSet{@ 1 @},
    GSet{@ 3, 7, 9, 11, 15 @},
    GSet{@ 2, 5, 4, 14, 16, 6, 13, 12, 8, 10 @}
]
> exists(orbit){x : x in Orbits(S1) | #x eq 5};
true
> orbit;
GSet{@ 3, 7, 9, 11, 15 @}
> IsSymmetric(ActionImage(S1, orbit));
true
```

So the stabilizer of the vertex 1 is Sym(5), acting faithfully on the 5 neighbours of vertex 1.

162.23 Symmetry and Regularity Properties of Graphs

`IsTransitive(G): GrphUnd -> BoolElt`

`IsVertexTransitive(G): GrphUnd -> BoolElt`

Returns `true` if the automorphism group of the graph G is transitive, otherwise `false`. To see which automorphism group is computed see Subsection *Graph Colouring and Automorphism Group*.

`IsEdgeTransitive(G): GrphUnd -> BoolElt`

Returns `true` if the automorphism group of the graph G is transitive on the edges of G (i.e. if the edge group of G is transitive). To see which automorphism group is computed see Subsection *Graph Colouring and Automorphism Group*.

`OrbitsPartition(G): GrphUnd -> [ { GrphVert} ]`

Given a graph G, return the partition of its vertex-set corresponding to the orbits of its automorphism group in the form of a set system. To see which automorphism group is computed see Subsection *Graph Colouring and Automorphism Group*.

`IsPrimitive(G): GrphUnd -> BoolElt`

Returns `true` if the graph G is primitive, i.e. if its automorphism group is primitive. To see which automorphism group is computed see Subsection *Graph Colouring and Automorphism Group*.

`IsSymmetric(G): GrphUnd -> BoolElt`

Returns `true` if the graph G is symmetric, i.e. if for all pairs of vertices u, v and w, t such

that u adj$v$ and w adj$t$, there exists an automorphism a such $u^a = w$ and $v^a = t$. To see which automorphism group is computed see Subsection *Graph Colouring and Automorphism Group*.

`IsDistanceTransitive(G): GrphUnd -> BoolElt`

Returns `true` if the connected graph G is distance transitive i.e. if for all vertices u, v, w, t of G such that $d(u, v) = d(w, t)$, there is an automorphism a in A such that $u^a = w$ and $v^a = t$. To see which automorphism group is computed see Subsection *Graph Colouring and Automorphism Group*.

`IsDistanceRegular(G): GrphUnd -> BoolElt`

Returns `true` if the graph G is distance regular, otherwise `false`. To see how the automorphism group of G is computed see Subsection *Graph Colouring and Automorphism Group*.

`IntersectionArray(G): GrphUnd -> [RngIntElt]`

The intersection array of the distance regular graph G. This is returned as a sequence $[k, b(1), \ldots, b(d-1), 1, c(2), \ldots, c(d)]$ where k is the valency of the graph, d is the diameter of the graph, and the numbers $b(i)$ and $c(i)$ are defined as follows: Let $N_j(u)$ denote the set of vertices of G that lie at distance j from vertex u. Let u and v be a pair of vertices satisfying $d(u, v) = j$.

Then $c(j)$ = number of vertices in $N_{j-1}(v)$ that are adjacent to u, $(1 \leq j \leq d)$, and $b(j)$ = number of vertices in $N_{j+1}(v)$ that are adjacent to u $(0 \leq j \leq d-1)$.

Example: Symmetry of the 8-Cube Graph `(ex-2c631b)`

We illustrate the use of some of the symmetry functions by applying then to the graph of the 8-dimensional cube.

```
> g := KCubeGraph(8);
> IsVertexTransitive(g);
true
> IsEdgeTransitive(g);
true
> IsSymmetric(g);
true
> IsDistanceTransitive(g);
true
> IntersectionArray(g);
[ 8, 7, 6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6, 7, 8 ]
```

We also see that the functions using the graph's automorphism group are dependent upon the graph being coloured or not:

```
> q := 9;
> P := FiniteProjectivePlane(q);
> X := IncidenceGraph(P);
>
> Order(X);
182
> Valence(X);
10
> Diameter(X);
3
> Girth(X);
6
> O1 := OrbitsPartition(X);
> IsSymmetric(X);
true
>
> Labels := [ "a" : i in [1..96] ];
```

(continues on next page)

(continued from previous page)

```
> #Labels;
96
> AssignLabels(VertexSet(X), Labels);
> O2 := OrbitsPartition(X);
> O2 eq O1;
false
> IsSymmetric(X);
false
```

162.24 Graph Databases and Graph Generation

Magma provides interfaces to some databases of certain graphs of interest. These databases are not provided by default with Magma, but may be downloaded from the optional databases section of the Magma website.

162.24.1 Strongly Regular Graphs

A catalogue of strongly regular graphs is available. This catalogue has been put together from various sources by B. McKay and can be found at

`http://cs.anu.edu.au/~bdm/data/`

Graphs in the database are indexed by a sequence of four parameters. They are, in order: the order of the graph, its degree, the number of common neighbours to each pair of adjacent vertices, and the number of common neighbours to each pair of non-adjacent vertices.

`StronglyRegularGraphsDatabase() -> DB`

Opens the database of strongly regular graphs.

`Classes(D): DB -> SeqEnum`

Returns all the parameter sequences used to index the graphs in the database D.

`NumberOfClasses(D): DB -> RngIntElt`

Returns the number of "classes" of graphs in the database D.

`NumberOfGraphs(D): DB -> RngIntElt`

Returns the number of graphs in the database D.

`NumberOfGraphs(D, S): DB, SeqEnum -> RngIntElt`

Returns the number of graphs in the database D with parameter sequence S.

`Graphs(D, S): DB, SeqEnum -> SeqEnum`

Returns (in a sequence) all the graphs in the database D with parameter sequence S.

`Graph(D, S, i): DB, SeqEnum, RngIntElt -> GrphUnd`

Returns the ith graph in the database D with parameter sequence S.

`RandomGraph(D): DB -> GrphUnd`

Returns a random graph in the database D.

`RandomGraph(D, S): DB, SeqEnum -> GrphUnd`

Returns a random graph in the database D with parameter sequence S.

`for G in D do ... end for;`

The database of strongly regular graphs may appear as the range in the `for`-statement.

Example: Database of Strongly Regular Graphs `(ex-a88364)`

The following few statements illustrate the basic access functions to the database of strongly regular graphs.

```
> D :=  StronglyRegularGraphsDatabase();
> Cs := Classes(D);
> Cs;
[
  [ 25, 8, 3, 2 ],
  [ 25, 12, 5, 6 ],
  [ 26, 10, 3, 4 ],
  [ 27, 10, 1, 5 ],
  [ 28, 12, 6, 4 ],
  [ 29, 14, 6, 7 ],
  [ 35, 16, 6, 8 ],
  [ 35, 14, 4, 6 ],
  [ 36, 15, 6, 6 ],
  [ 37, 18, 8, 9 ],
  [ 40, 12, 2, 4 ]
]
> assert NumberOfClasses(D) eq #Cs;
>
> NumberOfGraphs(D);
43442
>
> for i in [1..#Cs] do
>     NumberOfGraphs(D, Cs[i]);
> end for;
1
15
10
1
4
41
3854
180
32548
6760
28
>
> gs := Graphs(D, Cs[2]);
>
> g :=  Graph(D, Cs[2], Random(1, NumberOfGraphs(D, Cs[2])));
```

162.24.2 Small Graphs

An enumeration of small graphs with various properties has been created by B. McKay and may be found at

`http://cs.anu.edu.au/~bdm/data/graphs.html`

Certain of these databases are available from within Magma: Simple graphs, Eulerian graphs, planar connected graphs, and self-complementary graphs.

Creation of Small Graph Databases

`SmallGraphDatabase(n : parameters): RngIntElt -> DB`

```
IncludeDisconnected: Bool                    Default: false
```

Opens the database of simple graphs with n vertices, $2 \leq n \leq 10$. If the optional parameter `IncludeDisconnected` is set to true then the database will also include non-connected graphs.

`EulerianGraphDatabase(n : parameters): RngIntElt -> DB`

```
IncludeDisconnected: Bool                    Default: false
```

Opens the database of Eulerian graphs with n vertices, $3 \leq n \leq 11$. If the optional parameter `IncludeDisconnected` is set to true then the database will also include non-connected graphs. The allowed range of n for non-connected graphs is $2 \leq n \leq 12$.

`PlanarGraphDatabase(n): RngIntElt -> DB`

Opens the database of planar connected graphs with n vertices, $2 \leq n \leq 11$.

`SelfComplementaryGraphDatabase(n): RngIntElt -> DB`

Opens the database of self-complementary graphs with n vertices, $n \in \{4, 5, 8, 9, 12, 13, 16, 17, 20\}$. For $n = 20$ this is not a complete enumeration.

Access functions

`# D: DB -> RngIntElt`

Returns the number of graphs in the database D.

`Graph(D, i): DB, RngIntElt -> GrphUnd`

Returns the ith graph in the database D.

`Random(D): DB -> GrphUnd`

Returns a random graph from the database D.

`for G in D do ... end for;`

A database of small graphs may appear as the range in the `for`-statement.

162.24.3 Generating Graphs

We provide an interface to a graph generation programme, also due to B. McKay (see [McK98]). For the time being, users wanting to benefit from this facility must download the generation programme themselves directly from

`http://cs.anu.edu.au/~bdm/nauty/`

Important restriction: Since this program runs within a Unix pipe it is only available to users running Magma on a Unix platform.

And a note of caution: When the graph generation programme is used to generate reasonably large graphs ($n > 17$) it can be observed that the procedure of closing down the pipe (ie. closing the stream) may take some time. This will happen when the closing down attempt is made before the programme has completed the generation of all the graphs.

`GenerateGraphs(n : parameters): RngIntElt -> IO`

```
FirstGraph    : RngIntElt                    Default: 1
MinEdges      : RngIntElt                    Default:
MaxEdges      : RngIntElt                    Default:
Classes       : RngIntElt                    Default: 1
Class         : RngIntElt                    Default: 1
Connected     : BoolElt                      Default: false
Biconnected   : BoolElt                      Default: false
TriangleFree  : BoolElt                      Default: false
FourCycleFree: BoolElt                       Default: false
Bipartite     : BoolElt                      Default: false
MinDeg        : RngIntElt                    Default:
MaxDeg        : RngIntElt                    Default:
Canonical     : BoolElt                      Default: false
SparseRep     : BoolElt                      Default: false
```

Opens a pipe to the graph generation programme to generate all graphs of order n. Only available on Unix platforms.

The `GenerateGraphs` intrinsic allows the user to drive the generation programme via a set of parameters. These parameters are described below.

Once the generation programme has been downloaded from

`http://cs.anu.edu.au/~bdm/nauty/`

and compiled (using `make geng`), the resulting executable (named `geng` – it is *compulsory* that the resulting executable's name be `geng`) can be placed anywhere in the user's directory tree. The environment variable `MAGMA_NAUTY` *must* then be set to the path where the executable/command `geng` is to be found.

Reading of the generated graphs starts at the `FirstGraph`-th graph.

Generate graphs with minimum number of edges `MinEdges`.

Generate graphs with maximum number of edges `MaxEdges`.

Divide the generated graphs into disjoint `Classes` classes of very approximately equal size.

When generated graphs are divided into disjoint `Classes` classes, write only the `Class`th class.

Only generate connected graphs.

Only generate biconnected graphs.

Only generate triangle–free graphs.

Only generate 4–cycle–free graphs.

Only generate bipartite graphs.

Specify a lower bound for the minimum degree.

Specify an upper bound for the maximum degree.

Canonically label output graphs.

If `true`, generate the graphs in Sparse6 format (see below).

`NextGraph(I: parameters): IO -> BoolElt, GrphUnd`

`SparseRep: Bool` `Default: false`

Returns `true` if and only if I/O object I is not at the end of input. In this case the next graph is returned as well.

The graphs in I *must* be in either of the output formats Graph6 or Sparse6. Details on the Graph6 and Sparse6 format can be found at

`http://cs.anu.edu.au/~bdm/data/formats.html`

If SparseRep is `true` then the resulting graph will have a sparse representation. This of course is of special interest if the graphs read from I are also in Sparse6 format.

Example: Graph Generation (ex-56c803)

The following statements should help clarify the usage of the graph generation programme.

```
> I := GenerateGraphs (12:
>     FirstGraph:= 10,
>     Connected:= true,
>     Biconnected:= true,
>     TriangleFree:= true,
>     FourCycleFree:= true,
>     Bipartite:= true,
```

(continues on next page)

(continued from previous page)

```
>     MinDeg:= 1,
>     MaxDeg:= 9
> );
```

We'll read all the graphs from the 10th graph onwards (one can check that 28 graphs have been generated):

```
> count := 0;
> while true do
>     more := NextGraph(I);
>     if more then
>       count +:= 1;
>     else
>       break;
>     end if;
> end while;
> count;
19
```

If one wants to work with sparse graphs, it is recommended to proceed as follows:

```
> I := GenerateGraphs (6: SparseRep := true);
> count := 0;
> while true do
>     more := NextGraph(I: SparseRep := true);
>     if more then
>       count +:= 1;
>     else
>       break;
>     end if;
> end while;
> count;
156
```

162.24.4 A General Facility

In order to give users more flexibility in dealing with certain graph files the intrinsic `OpenGraphFile` is provided. It allows one to open either a graph file or a pipe to a graph generation programme. Since in both cases (file or Unix pipe) the outcome is the access to a stream of graphs, we henceforth refer to the graphs to be read as a *graph stream*.

The usual restriction: The `OpenGraphFile` which opens a pipe is only available to users running Magma on a Unix platform.

Accessing and reading the graph stream: Reading the graph stream is achieved by the above described access function *NextGraph*. As mentioned there, the graphs in the graph stream *must* be in either of the output formats Graph6 or Sparse6. This is why `OpenGraphFile` is restricted to streams of graphs in format Graph6 or Sparse6.

Details on the Graph6 and Sparse6 format can be found at

`http://cs.anu.edu.au/~bdm/data/formats.html`.

`OpenGraphFile(s, f, p): MonStgElt, RngIntElt, RngIntElt -> IO`

Opens a graph file or pipe at position p. If the stream to be opened is a Unix pipe then the string s *must* have the format "cmd command" where command stands for the command to run including necessary parameters. If the stream to be opened is a file the string s has format "filename".

The integer f indicates that the record length is fixed, which is true for streams in Graph6 format with every graph having the same order. This permits rapid positioning to position p in that case.

If in doubt, use $f = 0$. Also, positioning to 0 or positioning to 1 has the same effect of positioning to the start of the stream.

Opening a pipe is only available on Unix platforms. The I/O object I *must* contain graphs in Graph6 and Sparse6 format.

Example: Reading a Graph File (ex-49cbde)

As an example one could download one of the files found at

`http://cs.anu.edu.au/~bdm/data/`

Assuming this has been done, one can then proceed to read the graphs in the file:

```
> I := OpenGraphFile("/home/paule/graph/bdm_data/sr251256.g6", 0, 0);
>
> count := 0;
> more, g := NextGraph(I);
> while more do
>    count +:= 1;
>    more, g := NextGraph(I);
> end while;
> count;
15
```

Alternatively one could also drive the graph generation programme (or any other suitable programme for that matter) described in *Generating Graphs* using the `OpenGraphFile` access function.

The graph generation programme's name is `geng` (which can be found at

`http://cs.anu.edu.au/~bdm/nauty/`)

and has a help facility:

```
> I := OpenGraphFile("cmd /home/paule/graph/bdm_pgr/nauty/geng -help", 0, 0);

Usage: geng [-cCmtfbd#D#] [-uygsnh] [-lvq] [-x#X#] n [mine[:maxe]] [res/mod]
[file]

 Generate all graphs of a specified class.

      n    : the number of vertices (1..32)
 mine:maxe : a range for the number of edges
              #:0 means '# or more' except in the case 0:0
   res/mod : only generate subset res out of subsets 0..mod-1

     -c    : only write connected graphs
     -C    : only write biconnected graphs
     -t    : only generate triangle-free graphs
     -f    : only generate 4-cycle-free graphs
     -b    : only generate bipartite graphs
                 (-t, -f and -b can be used in any combination)
     -m    : save memory at the expense of time (only makes a
                 difference in the absence of -b, -t, -f and n <= 30).
     -d#   : a lower bound for the minimum degree
     -D#   : a upper bound for the maximum degree
     -v    : display counts by number of edges
     -l    : canonically label output graphs

     -u    : do not output any graphs, just generate and count them
     -g    : use graph6 output (default)
     -s    : use sparse6 output
```

(continues on next page)

(continued from previous page)

```
    -y    : use the obsolete y-format instead of graph6 format
    -h    : for graph6 or sparse6 format, write a header too

    -q    : suppress auxiliary output (except from -v)

  See program text for much more information.
```

Finally, here is a typical run of this graph generation programme:

```
> I := OpenGraphFile(
> "cmd /home/paule/graph/bdm_pgr/nauty/geng 15 -cCtfb -v", 0, 0);
>A geng -Ctfbd2D14 n=15 e=15-22
>C 4 graphs with 16 edges
>C 45 graphs with 17 edges
>C 235 graphs with 18 edges
>C 294 graphs with 19 edges
>C 120 graphs with 20 edges
>C 13 graphs with 21 edges
>C 1 graphs with 22 edges
>Z 712 graphs generated in 1.38 sec
```

162.25 Bibliography

- [AMO93] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows, Theory, Algorithms, and Applications*. Prentice Hall, 1993.
- [BM01] J. Boyer and W. Myrvold. Simplified O(n) Planarity Algorithms. submitted, 2001.
- [Bre79] D. Brelaz. New Methods to Color the Vertices of a Graph. *Communications of the ACM*, 22(9):251–256, 1979.
- [BK73] C. Bron and J. Kerbosch. Finding All Cliques of an Undirected Graph. *Communications of the ACM 9*, 16(9):575–577, 1973.
- [Chr75] N. Christofides. *Graph Theory, An Algorithm Approach*. Academic Press, 1975.
- [Eve79] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.
- [GM01] C. Gutwenger and P. Mutzel. A Linear Time Implementation of SPQR-Trees. In J. Marks, editor, *Graph Drawing 2000*, volume 1984 of LNCS, pages 70–90. Springer-Verlag, 2001.
- [HT73] J.E. Hopcroft and R.E. Tarjan. Dividing a Graph into Triconnected Components. *SIAM J. Comput.*, 2(3):135–158, 1973.
- [Lor89] P. Lorimer. The construction of Tutte's 8-cage and the Conder graph. *J. of Graph Theory*, 13(5):553–557, 1989.
- [McK81] B. D. McKay. Practical Graph Isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
- [McK98] B. D. McKay. Isomorph-free exhaustive generation. *J. Algorithms*, 26:306–324, 1998.
- [MP] B. D. McKay and A. Piperno. nauty and Traces User's Guide (Version 2.6). http://pallini.di.uniroma1.it/nug26.pdf.
- [MPW] W. Myrvold, T. Prsa, and N. Walker. A Dynamic Programming Approach for Timing and Designing Clique Algorithms. available at http://www.csr.uvic.ca/ wendym/.
- [TCR90] C.E. Leiserson T.H. Cormen and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

Chapter 163

Multigraphs

163.1 Introduction

Multigraphs and *multidigraphs* are graphs and digraphs which may have multiple (i.e., parallel) edges and loops. This is in contrast to simple graphs (see Section *Introduction* in the chapter on graphs). In the context of this chapter we use the term "graph" as a generic term for the general vertex-edge incident structure, and the term "multigraph" as a generic term whenever we want to emphasize the possible existence of multiple edges and loops. Thus, graphs and multigraphs may be directed or undirected. The meaning of the terms "graph" and "multigraph" should be made clear from the context in which they occur.

Multigraphs are represented in the form of an adjacency list; for more information on this topic we refer the reader to Section *Graphs with a Sparse Representation*. As is the case for simple graphs, the vertices and edges of a multigraph may be given "decorations". More precisely, they may be labelled; in addition the edges may also be assigned a capacity (if one is interested in flow problems) and/or a weight (for investigation of shortest path problems).

For convenience, Magma provides users with *networks*: in the Magma context, a network is understood as being a multidigraph whose edges are *always* given a capacity. Any Magma function that takes a network as an argument will usually take any graph whose edges have a capacity as an argument. Networks are covered in Chapter *Networks*; a few functionalities specific to networks will, however, be discussed in the general multigraph context whenever there is a need to highlight some differences between networks and general multi(di)graphs.

In particular, almost all the standard graph construction functions (see Section *Standard Construction for Multigraphs*) preserve the graph's support set and vertex and edge decorations. That is, the resulting graph will have a support and vertex and edge decorations compatible with the original graph and the operation performed on that graph.

A Magma multigraph object has type `GrphMultUnd`, a multidigraph has type `GrphMultDir`, and a network has type `GrphNet`. All three objects are of type `GrphMult`.

The multigraph facilities represent a significant subset of the (simple) graph functionality. Consequently, some sections of this chapter are very similar to their counterparts in Chapter *Graphs*. For the sake of clarity, this chapter describes the complete multigraph functionality.

163.2 Construction of Multigraphs

In this implementation, the order n of a multigraph or multidigraph is bounded by 134217722. See Section *Bounds on the Graph Order* in Chapter *Graphs* for more details.

163.2.1 Construction of a General Multigraph

Undirected multigraphs are constructed in a similar way to graphs (Subsection *Construction of a General Graph*).

`MultiGraph<n | edges >: RngIntElt, List -> GrphMultUnd, GrphVertSet, GrphEdgeSet`

`MultiGraph<S | edges >: SetEnum, List -> GrphMultUnd, GrphVertSet, GrphEdgeSet`

`MultiGraph<S | edges >: SetIndx, List -> GrphMultUnd, GrphVertSet, GrphEdgeSet`

Construct the multigraph G with vertex-set $V = \{@v_1, v_2, \ldots, v_n@\}$ (where $v_i = i$ for each i if the first form of the constructor is used, or the ith element of the enumerated or indexed set S otherwise), and edge-set $E = \{e_1, e_2, \ldots, e_q\}$. This function returns three values: The multigraph G, the vertex-set V of G; and the edge-set E of G.

The elements of E are specified by the list *edges*, where the items of *edges* may be objects of the following types:

(a) A pair $\{v_i, v_j\}$ of vertices in V. The undirected edge $\{v_i, v_j\}$ from v_i to v_j will be added to the edge-set for G.

(b) A tuple of the form $\langle v_i, N_i\rangle$ where N_i will be interpreted as a set of neighbours for the vertex v_i. The elements of the sets N_i must be elements of V. If $N_i = \{u_1, u_2, \ldots, u_r\}$, the edges $\{v_i, u_1\}, \ldots, \{v_i, u_r\}$ will be added to G.

(c) A sequence $[N_1, N_2, \ldots, N_n]$ of n sets, where N_i will be interpreted as a set of neighbours for the vertex v_i. The edges $\{v_i, u_i\}$, $u_i \in N_i$, are added to G.

In addition to these three basic ways of specifying the *edges* list, the items in *edges* may also be:

(d) An edge e of a graph or digraph or multigraph or multidigraph or network of order n. If e is an edge from u to v, then the edge $\{u, v\}$ is added to G.

(e) An edge-set E of a graph or digraph or multigraph or multidigraph or network of order n. Every edge e in E will be added to G according to the rule set out for a single edge.

(f) A graph or a digraph or a multigraph or a multidigraph or a network H of order n. Every edge e in H's edge-set is added to G according to the rule set out for a single edge.

(g) A set of

(i) Pairs of the form $\{v_i, v_j\}$ of vertices in V.

(ii) Tuples of the form $\langle v_i, N_i\rangle$ where N_i will be interpreted as a set of neighbours for the vertex v_i.

(iii) Edges of a graph or digraph or multigraph or multidigraph or network of order n.

(iv) Graphs or digraphs or multigraphs or multidigraphs or networks of order n.

(h) A sequence of

(i) Tuples of the form $\langle v_i, N_i\rangle$ where N_i will be interpreted as a set of neighbours for the vertex v_i.

Example: Constructing a Multigraph `(ex-3c3311)`

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multigraph
Vertex  Neighbours

1       2 3 2 ;
2       3 2 2 1 1 ;
3       2 1 ;
```

163.2.2 Construction of a General Multidigraph

Multidigraphs are constructed in the same way as digraphs (Subsection *Construction of a General Digraph*).

`MultiDigraph<n | edges >: RngIntElt, List -> GrphMultDir, GrphVertSet, GrphEdgeSet`

`MultiDigraph<S | edges >: SetEnum, List -> GrphMultDir, GrphVertSet, GrphEdgeSet`

`MultiDigraph<S | edges >: SetIndx, List -> GrphMultDir, GrphVertSet, GrphEdgeSet`

Construct the multidigraph G with vertex-set $V = \{@v_1, v_2, \ldots, v_n@\}$ (where $v_i = i$ for each i if the first form of the constructor is used, or the ith element of the enumerated or indexed set S otherwise), and edge-set $E = \{e_1, e_2, \ldots, e_q\}$. This function returns three values: The multidigraph G, the vertex-set V of G; and the edge-set E of G.

The elements of E are specified by the list *edges*, where the items of *edges* may be objects of the following types:

(a) A pair $[v_i, v_j]$ of vertices in V. The directed edge $[v_i, v_j]$ from v_i to v_j will be added to the edge-set for G.

(b) A tuple of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i. The elements of the sets N_i must be elements of V. If $N_i = \{u_1, u_2, \ldots, u_r\}$, the edges $[v_i, u_1], \ldots, [v_i, u_r]$ will be added to G.

(c) A sequence $[N_1, N_2, \ldots, N_n]$ of n sets, where N_i will be interpreted as a set of out-neighbours for the vertex v_i. All the edges $[v_i, u_i]$, $u_i \in N_i$, are added to G.

In addition to these four basic ways of specifying the *edges* list, the items in *edges* may also be:

(d) An edge e of a graph or digraph or multigraph or multidigraph or network of order n. If e is an edge from u to v, then the edge $[u, v]$ is added to G. Thus, if e is an undirected edge from u to v, both edges $[u, v]$ and $[v, u]$ are added to G.

(e) An edge-set E of a graph or digraph or multigraph or multidigraph or network of order n. Every edge e in E will be added to G according to the rule set out for a single edge.

(f) A graph or a digraph or a multigraph or a multidigraph or a network H of order n. Every edge e in H's edge-set is added to G according to the rule set out for a single edge.

(g) A set of

(i) Pairs of the form $[v_i, v_j]$ of vertices in V.

(ii) Tuples of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i.

(iii) Edges of a graph or digraph or multigraph or multidigraph or network of order n.

(iv) Graphs or digraphs or multigraphs or multidigraphs or networks of order n.

(h) A sequence of

(i) Tuples of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i.

Example: Constructing a Multidigraph `(ex-f50793)`

```
> G := MultiDigraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multidigraph
Vertex  Neighbours

1       2 3 2 ;
2       3 2 ;
3       ;
```

163.2.3 Printing of a Multi(di)graph

A multi(di)graph is displayed by listing, for each vertex, all of its adjacent vertices. If the multigraph has multiple edges from u to v, then the adjacency list of u contains as many copies of the vertex v as there are edges from u to v.

The vertices in the adjacency list are not ordered, they appear in the order in which they were created. See the previous examples *Constructing a Multigraph* and *Constructing a Multidigraph*.

163.2.4 Operations on the Support

The support of a multi(di)graph is subject to exactly the same operations as simple graphs (see Subsection *Operations on the Support*).

`Support(G): GrphMult -> SetIndx`

`Support(V): GrphVertSet -> SetIndx`

The indexed set used in the construction of G (or the graph for which V is the vertex-set), or the standard set $\{@1, \ldots, n@\}$ if it was not given.

`ChangeSupport(G, S): GrphMult, SetIndx -> GrphMult, GrphVertSet, GrphEdgeSet`

If G is a graph having n vertices and S is an indexed set of cardinality n, return a new graph H equal to G but whose support is S. That is, H is structurally equal to G and its vertex and edge decorations are the *same* as those for G (see Sections *Vertex Decorations: Labels* and *Edge Decorations*).

`ChangeSupport(~G, S): GrphMult, SetIndx`

The procedural version of the above function.

`StandardGraph(G): GrphMult -> GrphMult`

Returns a graph H that is isomorphic to G but defined on the standard support. That is, H is structurally equal to G and its vertex and edge decorations are the same as those for G.

Example: Support of a Multigraph `(ex-256013)`

```
> S := {@ "a", "b", "c" @};
> G := MultiGraph< S | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multigraph
Vertex  Neighbours

c       b a b ;
b       a b b c c ;
a       b c ;

> StandardGraph(G);
Multigraph
Vertex  Neighbours

1       2 3 2 ;
2       3 2 2 1 1 ;
3       2 1 ;
```

163.3 The Vertex–Set and Edge–Set of Multigraphs

Much of the functionality for simple graphs (see Section *The Vertex–Set and Edge–Set of a Graph*) also applies to multigraphs. We will not repeat here the functions pertaining to the vertex-set and edge-set of a graph, but concentrate instead on the edges. Indeed, there is a difference in the manner in which multigraph edges are created and accessed when compared to simple graph edges.

Since multigraphs may have multiple edges from u to v, it is necessary to know how many of these edges there are, and how each can be accessed. More importantly a scheme is required to uniquely identify an edge.

Recall that a multigraph is represented by means of an adjacency list, so any edge from u to v can be identified by its index in this list. If there is more than one edge from u to v, then a list of indices will identify each edge from u to v. Thus, the index of the edge in the adjacency list will be the means by which this edge can be *uniquely* identified.

This has a bearing on how an edge can be coerced into a multigraph: Successful coercion will require two vertices u and v so that v is a neighbour of u, and a valid index i in the multigraph's adjacency list. That is, the position i in the list is the index of an edge from u to v.

`EdgeIndices(u, v): GrphVert, GrphVert -> SeqEnum`

`Indices(u, v): GrphVert, GrphVert -> SeqEnum`

Given vertices u and v of a multigraph G, returns the indices of the possibly multiple edge from u to v.

`EdgeMultiplicity(u, v): GrphVert, GrphVert -> RngIntElt`

`Multiplicity(u, v): GrphVert, GrphVert -> RngIntElt`

Given vertices u and v of a multigraph G, returns the multiplicity of the possibly multiple edge from u to v. Returns 0 if u is not adjacent to v.

`Edges(u, v): GrphVert, GrphVert -> SeqEnum`

Given vertices u and v of a multigraph G, returns all the edges from u to v as a sequence of elements of the edge-set of G.

`IncidentEdges(u): GrphVert -> SetEnum`

Given a vertex u of an undirected multigraph G, returns all the edges incident to u as a set of elements of the edge-set of G. If G is a multidigraph, the function returns all the edges incident to *and* from u as a set of elements of the edge-set of G.

`E ! < { u, v }, i >: GrphEdgeSet, < . > -> GrphEdge`

Given the edge-set E of the undirected multigraph G and objects u, v belonging to the support of G corresponding to adjacent vertices, returns the edge from u to v which corresponds to the edge with index i in the adjacency list of G. This requires that the edge at i in the adjacency list of G is an edge from u to v.

`E ! < [ u, v ], i >: GrphEdgeSet, < . > -> GrphEdge`

Given the edge-set E of the multidigraph G and objects u, v belonging to the support of G corresponding to adjacent vertices, returns the edge from u to v which corresponds to the edge with index i in the adjacency list of G. This requires that the edge at i in the adjacency list of G is an edge from u to v.

`E.i {E . i}: GrphEdgeSet, RngIntElt -> GrphEdge`

Let E be the edge-set of G. If G is a simple graph (digraph) then this function is as described in Subsection *Creating Edges and Vertices*.

If G is a multigraph or multidigraph, then this function returns the edge at index i in the adjacency list of G, provided i is a valid index.

`EndVertices(e): GrphEdge -> { GrphVert, GrphVert }`

`EndVertices(e): GrphEdge -> [ GrphVert, GrphVert ]`

Given an edge e in an undirected multigraph G, returns the end vertices of e as a set of vertices $\{u, v\}$. If G is a multidigraph, returns e's end vertices as a sequence of vertices $[u, v]$.

`InitialVertex(e): GrphEdge -> GrphVert`

Given an edge $e = \{u, v\}$ or $e = [u, v]$, returns vertex u. This is useful in the undirected case since it indicates, where relevant, the direction in which the edge has been traversed.

`TerminalVertex(e): GrphEdge -> GrphVert`

Given an edge $e = \{u, v\}$ or $e = [u, v]$, returns vertex v. This is useful in the undirected case since it indicates, where relevant, the direction in which the edge has been traversed.

`Index(e): GrphEdge -> RngIntElt`

Given an edge e in a multi(di)graph G, returns the index of e in the adjacency list of G.

`s eq t: GrphEdge, GrphEdge -> BoolElt`

Returns `true` if the edge s is equal to the edge t. It s and t are edges in a multi(di)graph G, returns `true` if and only if s and t have the same index in the adjacency list of G.

Example: Edges of a Multigraph (ex-16c702)

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
```

The graph G has a loop at vertex 2:

```
> Edges(G);
{@ < {1, 2}, 1 >, < {1, 2}, 5 >, < {1, 3}, 3 >, < {2, 2}, 7 >, < {2,
3}, 9 > @}
> E := EdgeSet(G);
> E.7;
< {2, 2}, 7 >
> assert InitialVertex(E.7) eq TerminalVertex(E.7);
```

The graph G has a multiple edge from vertex 1 to vertex 2:

```
> u := VertexSet(G)!1;
> v := VertexSet(G)!2;
> I := EdgeIndices(u, v);
> I;
[ 5, 1 ]
> assert #I eq Multiplicity(u, v);
>
> E := EdgeSet(G);
> e1 := E!< { 1, 2 }, I[1] >;
> e2 := E!< { 1, 2 }, I[2] >;
> e1, e2;
< {1, 2}, 5 > < {1, 2}, 1 >
> EndVertices(e1);
{ 1, 2 }
> EndVertices(e2);
{ 1, 2 }
> assert Index(e1) eq I[1];
> assert Index(e2) eq I[2];
>
> assert e1 eq E.I[1];
> assert not e2 eq E.I[1];
> assert e2 eq E.I[2];
> assert not e1 eq E.I[2];
```

We have seen that these two edges can be accessed directly:

```
> Edges(u, v);
[ < {1, 2}, 5 >, < {1, 2}, 1 > ]
```

163.4 Vertex and Edge Decorations

163.4.1 Vertex Decorations: Labels

In this implementation, it is only possible to assign labels as vertex decorations.

A vertex labelling of a graph G is a partial map from the vertex-set of G into a set L of labels.

`AssignLabel(~G, u, l): GrphMult, GrphVert, .`

Assigns the label l to the vertex u in the graph G.

`AssignLabels(~G, S, L): GrphMult, [GrphVert], SeqEnum`

`AssignLabels(~G, S, L): GrphMult, {@ GrphVert @}, SeqEnum`

Assigns the labels in L to the corresponding vertices in G in the sequence or indexed set S. If for vertex u the corresponding entry in L is not defined then any existing label of u is removed.

`AssignVertexLabels(~G, L): GrphMult, SeqEnum`

Assigns the labels in L to the corresponding vertices in the graph G.

`IsLabelled(u): GrphVert -> BoolElt`

Returns `true` if and only if the vertex u has a label.

`IsLabelled(V): GrphVertSet -> BoolElt`

Returns `true` if and only if the vertex-set V is labelled.

`IsVertexLabelled(G): GrphMult -> BoolElt`

Returns `true` if and only if vertices of the graph G are labelled.

`Label(u): GrphVert -> .`

The label of the vertex u. An error is raised if u is not labelled.

`Labels(S): [GrphVert] -> SeqEnum`

The sequence L of labels of the vertices in the sequence S. If an element of S has no label, then the corresponding entry in L is undefined.

`Labels(V): GrphVertSet -> SeqEnum`

The sequence L of labels of the vertices in the vertex-set V. If an element of V has no label, then the corresponding entry in L is undefined.

`VertexLabels(G): GrphMult -> SeqEnum`

The sequence L of labels of the vertices of G. If a vertex of G has no label, then the corresponding entry in L is undefined.

`DeleteLabel(~G, u): GrphMult, GrphVert`

Removes the label of the vertex u.

`DeleteLabels(~G, S): GrphMult, [GrphVert]`

Remove the labels of the vertices in S.

`DeleteVertexLabels(~G): GrphMult`

Remove the labels of the vertices in the graph G.

163.4.2 Edge Decorations

Simple graph or multigraph edges can be assigned three different types of decorations:

- a label,
- a capacity,
- a weight.

Edge labels can be objects of any Magma type. Edge capacities must be non-negative integers (and loops must be assigned a zero capacity). Edge weights must be elements of a totally ordered ring. Not all edges need to be assigned labels when labelling edges, nor do all edges need to be assigned capacities or weights when assigning either decoration. In the latter case, if some edges have been assigned a capacity or a weight, then the capacity or weight of any remaining unassigned edge is always taken to be zero.

One may want to assign capacities to edges in order to apply a network-flow algorithm to the graph (Section *Maximum Flow and Minimum Cut*). By assigning weights to edges one may also be able to run a shortest-path algorithm (Section *Distances, Shortest Paths, and Minimum Weight Trees*).

Assigning Edge Decorations

`AssignLabel(~G, e, l): GrphMult, GrphEdge, .`

`AssignCapacity(~G, e, c): GrphMult, GrphEdge, RngIntElt`

`AssignWeight(~G, e, w): GrphMult, GrphEdge, RngElt`

Assigns the label l or the capacity c or the weight w to the edge e in the graph G. The capacity of any edge must be a non-negative integer except in the case of a loop when it must be zero. The weight w must be an element from a totally ordered ring.

`AssignLabels(~G, S, D): GrphMult, [GrphEdge], SeqEnum`

`AssignLabels(~G, S, D): GrphMult, {@ GrphEdge @}, SeqEnum`

`AssignCapacities(~G, S, D): GrphMult, [GrphEdge], [RngIntElt]`

`AssignCapacities(~G, S, D): GrphMult, {@ GrphEdge @}, [RngIntElt]`

`AssignWeights(~G, S, D): GrphMult, [GrphEdge], [RngElt]`

`AssignWeights(~G, S, D): GrphMult, {@ GrphEdge @}, [RngElt]`

Assigns the labels or capacities or weights in the sequence D to the corresponding edges in the sequence or indexed set S. If for some edge e the corresponding entry in D is not defined then any existing label or capacity or weight of e is removed. The same constraints regarding capacity and weight apply as in the single edge assignment case.

`AssignEdgeLabels(~G, D): GrphMult, SeqEnum`

`AssignCapacities(~G, D): GrphMult, [RngIntElt]`

`AssignWeights(~G, D): GrphMult, [RngElt]`

Assigns the labels or capacities or weights in the sequence D to the corresponding edges in graph G. Let E be the edge-set of G and let d be the decoration at position i in D, that is, $d = D[i]$. Then the corresponding edge e in E which will be decorated is $E.i$ (see `E.i {E . i}`).

If for some edge e the corresponding entry in D is not defined then any existing label or capacity or weight of e is removed. The same constraints regarding capacity and weight apply as in the single edge assignment case.

Testing for Edge Decorations

While it is the case that an edge is considered to be labelled if and only if it has been assigned a label, an edge may have a default capacity (weight) of zero. Let e be an edge of a graph G and assume that e has not been assigned a capacity (weight).

If any other edge of G has been assigned a capacity (weight), then the edge-set of G is considered to be capacitated (weighted). In which case the capacity (weight) of e has a default value of zero.

If no other edge of G has been assigned a capacity (weight), then the edge-set of G is considered to be uncapacitated (unweighted). In which case asking for the capacity (weight) of e results in an error.

This is in contrast to the labelling situation where there is no concept of a "default" label. The edge-set of G is considered to be labelled if and only if at least one edge of G has been labelled, while any edge of G is considered to be labelled if and only if e has been labelled.

`IsLabelled(e): GrphEdge -> BoolElt`

Returns `true` if and only if the edge e has a label.

`IsLabelled(E): GrphEdgeSet -> BoolElt`

Returns `true` if and only if the edge-set is labelled; that is, if and only if at least one edge of E has been assigned a label.

`IsEdgeLabelled(G): GrphMult -> BoolElt`

`IsEdgeLabelled(G): Grph -> BoolElt`

Returns `true` if and only if the edge-set of G is labelled.

`IsCapacitated(E): GrphEdgeSet -> BoolElt`

Returns `true` if and only if the edge-set is capacitated; that is, if and only if at least one edge of E has been assigned a capacity.

`IsEdgeCapacitated(G): GrphMult -> BoolElt`

`IsEdgeCapacitated(G): Grph -> BoolElt`

Returns `true` if and only if the edge-set of G is capacitated.

`IsWeighted(E): GrphEdgeSet -> BoolElt`

Returns `true` if and only if the edge-set is weighted; that is, if and only if at least one edge of E has been assigned a weight.

`IsEdgeWeighted(G): GrphMult -> BoolElt`

`IsEdgeWeighted(G): Grph -> BoolElt`

Returns `true` if and only if the edge-set of G is weighted.

Reading Edge Decorations

An edge may have a default capacity and weight of zero, see Subsubsection *Testing for Edge Decorations* for more details.

`Label(e): GrphEdge -> .`

The label of the edge e. An error is raised if e has not been assigned a capacity.

`Capacity(e): GrphEdge -> RngIntElt`

The capacity of the edge e. Let G be the parent graph of e. An error is raised if the edge-set of G is uncapacitated. If the edge-set of G is capacitated but e has not been assigned a capacity, then `Capacity(e)` returns zero as the default value.

`Weight(e): GrphEdge -> RngElt`

The weight of the edge e. Let G be the parent graph of e. An error is raised if the edge-set of G is unweighted. If the edge-set of G is weighted but e has not been assigned a weight, then `Weight(e)` returns zero as the default value.

`Labels(S): [GrphEdge] -> SeqEnum`

`Capacities(S): [GrphEdge] -> SeqEnum`

`Weights(S): [GrphEdge] -> SeqEnum`

The sequence D of labels or capacities or weights of the edges in the sequence S. Let E be the edge-set of the parent graph of the edges. If E is unlabelled or uncapacitated or unweighted then D is the null sequence. If an element of S has no label then the corresponding entry in D is undefined. If an element of S has no capacity or weight while E is capacitated or weighted then the corresponding entry in D has the default value of zero.

`Labels(E): GrphEdgeSet -> SeqEnum`

`Capacities(E): GrphEdgeSet -> SeqEnum`

`Weights(E): GrphEdgeSet -> SeqEnum`

The sequence D of labels or capacities or weights of the edges in the edge-set E. If E is unlabelled or uncapacitated or unweighted then D is the null sequence. If an element of E has no label then the corresponding entry in D is undefined. If an element of E has no capacity or weight while E is capacitated or weighted then the corresponding entry in D has the default value of zero. The corresponding entry i in D of any edge e is such that $e = E.i$ (see `E.i {E . i}`).

`EdgeLabels(G): GrphMult -> SeqEnum`

`EdgeLabels(G): Grph -> SeqEnum`

`EdgeCapacities(G): GrphMult -> SeqEnum`

`EdgeCapacities(G): Grph -> SeqEnum`

`EdgeWeights(G): GrphMult -> SeqEnum`

`EdgeWeights(G): Grph -> SeqEnum`

The sequence D of labels or capacities or weights of the edges in edge-set E of the graph G. If E is unlabelled or uncapacitated or unweighted then D is the null sequence. If an element of E has no label then the corresponding entry in D is undefined. If an element of E has no capacity or weight while E is capacitated or weighted then the corresponding entry in D has the default value of zero. The corresponding entry i in D of any edge e is such that $e = E.i$ (see `E.i {E . i}`).

Deleting Edge Decorations

`DeleteLabel(~G, e): GrphMult, GrphEdge`

`DeleteCapacity(~G, e): GrphMult, GrphEdge`

`DeleteWeight(~G, e): GrphMult, GrphEdge`

Removes the label or capacity or weight of the edge e in the graph G.

`DeleteLabels(~G, S): GrphMult, [GrphEdge]`

`DeleteCapacities(~G, S): GrphMult, [GrphEdge]`

`DeleteWeights(~G, S): GrphMult, [GrphEdge]`

Remove the labels or capacities or weights of the edges (of the graph G) in S.

`DeleteEdgeLabels(~G): GrphMult`

`DeleteCapacities(~G): GrphMult`

`DeleteWeights(~G): GrphMult`

Remove the labels or capacities or weights of the edges in the edge set of the graph G.

163.4.3 Unlabelled, or Uncapacitated, or Unweighted Graphs

Starting with a graph G, the functions below return a graph that is isomorphic as a simple graph to G, but without the vertex and edge labels of G, (or without the edge capacities or weights of G in the case of a network). Should one require a copy of a graph without the support of G, see Subsection *Operations on the Support*. Should one require a copy of a graph without the support of G *and* without the vertex/edge decorations of G, see Subsection *Converting between Simple Graphs and Multigraphs*.

`UnlabelledGraph(G): GrphMult -> GrphMult`

Return the (vertex and edge) unlabelled graph structurally identical to G, whose edges have the same capacities and weights as those in G. The support of G is also retained in the resulting graph.

`UncapacitatedGraph(G): GrphMult -> GrphMult`

Return the uncapacitated graph structurally identical to G, whose vertices and edges have the same labels, and whose edges have the same weights as those in G. The support of G is also retained in the resulting graph.

`UnweightedGraph(G): GrphMult -> GrphMult`

Return the unweighted graph structurally identical to G, whose vertices and edges have the same labels, and whose edges have the same capacities as those in G. The support of G is also retained in the resulting graph.

Example: Two-Colouring a Complete Bipartite Graph `(ex-51c6ce)`

The labelling operations are illustrated by constructing a 2-colouring of the complete bipartite graph $K_{3,4}$. Use is made of the function `Distance`(u, v) which returns the distance between vertices u and v.

```
> K34, V, E := BipartiteGraph(3, 4);
> L := [ IsEven(Distance(V!1, v)) select "red" else "blue" : v in Vertices(K34)␣
↪];
> AssignLabels(Vertices(K34), L);
> VertexLabels(K34);
[ red, red, red, blue, blue, blue, blue ]
```

Example: Cayley Graph of a Group (ex-c76dcd)

Another illustration is the creation of the Cayley graph of a group. In this example Sym(4) is used.

```
> G<a,b> := FPGroup(Sym(4));
> I, m := Transversal(G, sub<G | 1>);
> S := Setseq(Generators(G));
> N := [ {m(a*b) : b in S} : a in I ];
> graph := StandardGraph(Digraph< I | N >);
> AssignLabels(VertexSet(graph), IndexedSetToSequence(I));
> V := VertexSet(graph);
> E := EdgeSet(graph);
> for i in [1..#I] do
>     AssignLabels([ E![V | i, Index(I, m(I[i]*s))] : s in S], S);
> end for;
```

In this graph, [1,2,5,4] is a cycle. So the corresponding edge labels should multiply to the identity.

```
> &*Labels([ EdgeSet(graph) | [1,2], [2,5], [5,4], [4,1] ]);
a^4
> G;
Finitely presented group G on 2 generators
Relations
    b^2 = Id(G)
    a^4 = Id(G)
    (a^-1 * b)^3 = Id(G)
```

Example: Edge Decorations of a Multigraph (ex-1a5a8b)

We turn to multigraphs to illustrate a point with respect to the listing of the edge decorations. We assign random labels, capacities, and weights to a multigraph.

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> E := EdgeSet(G);
> I := Indices(G);
>
> for i in I do
>     AssignLabel(~G, E.i, Random([ "a", "b", "c", "d" ]));
>     if not InitialVertex(E.i) eq TerminalVertex(E.i) then
>         AssignCapacity(~G, E.i, Random(1, 3));
>     end if;
>     AssignWeight(~G, E.i, Random(1, 3));
> end for;
>
> EdgeLabels(G);
[ c, undef, c, undef, d, undef, c, undef, c ]
> EdgeCapacities(G);
[ 2, undef, 3, undef, 2, undef, 0, undef, 3 ]
> EdgeWeights(G);
[ 3, undef, 2, undef, 1, undef, 2, undef, 1 ]
```

Since G is undirected, the edge $\{v, u\}$ and the edge $\{v, u\}$ are the same object, and thus they have the same index:

```
> V := VertexSet(G);
> u := V!1;
> v := V!3;
> Indices(u, v);
[ 3 ]
> Indices(v, u);
[ 3 ]
```

However, since G is represented by means of an adjacency list, the undirected edge $\{u, v\}$ is stored twice in the list, and so there are two positions in the list associated with the edge. By convention, these positions are contiguous, but, more importantly from the user's perspective, the function `Index` that returns the index of the edge $\{u, v\}$ *always* returns the odd index associated with the edge. This explains why, for an undirected multigraph, the sequence returned by a function like `EdgeLabels` will always have undefined elements.

Finally note that the loop $\{2, 2\}$, which was assigned no capacity, is shown to have capacity zero:

```
> E := EdgeSet(G);
> E.7;
< {2, 2}, 7 >
> Capacity(E.7);
0
```

This is in accordance with the definition of the default value for edge capacity and weight (see Subsubsection *Testing for Edge Decorations*).

163.5 Standard Construction for Multigraphs

As noted in the Introduction *Introduction*, most of the functions listed in this section correctly handle a graph's support and vertex/edge decorations. That is, these attributes are inherited by the graph created as a result of applying such a function.

163.5.1 Subgraphs

The construction of subgraphs from multigraphs or multidigraphs is similar to the construction of subgraphs from simple graphs or digraphs (see Subsection *Subgraphs and Quotient Graphs*).

Note that the support set, vertex labels, and edge decorations are transferred from the supergraph to the subgraph.

`sub< G | list >: GrphMult, List -> GrphMult, GrphVertSet, GrphEdgeSet`

Construct the multigraph H as a subgraph of G. The function returns three values: The multigraph H, the vertex-set V of H; and the edge-set E of H. If G has a support set and/or if G has vertex/edge labels, and/or edge capacities or edge weights then *all* these attributes are transferred to the subgraph H.

The elements of V and of E are specified by the list *list* whose items can be objects of the following types:

(a) A vertex of G. The resulting subgraph will be the subgraph induced on the subset of `VertexSet(G)` defined by the vertices in *list*.

(b) An edge of G. The resulting subgraph will be the subgraph with vertex-set `VertexSet(G)` whose edge-set consists of the edges in *list*.

(c) A set of

(i) Vertices of G.

(ii) Edges of G.

If the list of vertices and edges happens to contain duplicate elements, they will be ignored by the subgraph constructor. It is easy to recover the map that sends the vertices of the subgraph to the vertices of the supergraph and vice-versa: Simply coerce the vertex of the subgraph into the supergraph's vertex-set, and, if applicable, coerce the vertex of the supergraph into the subgraph's vertex-set.

Example: Subgraphs of a Decorated Multidigraph (ex-d47089)

We create a multidigraph G with a support set; we assign labels to its vertices, and labels, capacities, and weights to its edges. This demonstrates the fact that any subgraph of G retains the support set, as well as the vertex and edge decorations.

```
> S := {@ "a", "b", "c" @};
> G := MultiDigraph< S | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multidigraph
Vertex  Neighbours

a       b c b ;
b       c b ;
c       ;

>
> V := VertexSet(G);
> for u in V do
>     AssignLabel(~G, u, Random([ "X", "Y", "Z" ]));
> end for;
>
> E := EdgeSet(G);
> I := Indices(G);
> for i in I do
>     AssignLabel(~G, E.i, Random([ "D", "E", "F" ]));
>     if not InitialVertex(E.i) eq TerminalVertex(E.i) then
>         AssignCapacity(~G, E.i, Random(1, 3));
>     end if;
>     AssignWeight(~G, E.i, Random(1, 3));
> end for;
>
> VertexLabels(G);
[ Z, Y, Z ]
> EdgeLabels(G);
[ E, F, D, E, E ]
> EdgeCapacities(G);
[ 2, 1, 3, 0, 2 ]
> EdgeWeights(G);
[ 2, 1, 3, 1, 1 ]
>
> V := VertexSet(G);
> H := sub< G | V!1, V!2 >;
> H;
Multidigraph
Vertex  Neighbours

a       b b ;
b       b ;
```

(continues on next page)

(continued from previous page)

```
>
> for u in VertexSet(H) do
>     assert Label(u) eq Label(V!u);
> end for;
>
> for e in EdgeSet(H) do
>     u := InitialVertex(e);
>     v := TerminalVertex(e);
>
>     assert SequenceToSet(Labels(Edges(u, v)))
>     eq SequenceToSet(Labels(Edges(V!u, V!v)));
>     assert SequenceToSet(Capacities(Edges(u, v)))
>     eq SequenceToSet(Capacities(Edges(V!u, V!v)));
>     assert SequenceToSet(Weights(Edges(u, v)))
>     eq SequenceToSet(Weights(Edges(V!u, V!v)));
> end for;
```

Note that since G is a multidigraph it is not possible to coerce an edge of H into the edge-set of G. This is because edge coercion for multi(di)graphs involves the coercion of both the end-vertices *and* the index of the edge.

A correspondence is established between the edges of H and the edges of G by retrieving *all* the edges in H and G having same end-vertices.

163.5.2 Multigraphs Constructed from Groups

`CayleyMultiGraph(A : parameter): Grp -> Grph, GrphVertSet, GrphEdgeSet`

```
Labelled: BoolElt                    Default: true
Directed: BoolElt                    Default: true
GenSet  : SetEnum                    Default:
```

Given a finite group A defined on generating set X, construct the Cayley multigraph C of A relative to the generating set X. This graph is defined as follows: The vertices correspond to the elements of A and, for each x in X and each element u of A, there is an edge from u to $u*x$. When `Directed` is `false` and x is an involution (so $x^2 = 1$), the unordered pair $\{u, u * x\}$ arises from both u and $u * x$, so, unlike `CayleyGraph`, *two* parallel edges are produced between them rather than being merged into a single edge.

The optional parameter `Labelled` (`Labelled := true` by default) can be set to `false` to prevent the graph being labelled. If this is not done, then the vertices of C will be labelled with the appropriate elements of A and the (directed) edge from u to v will be labelled with the appropriate element x as defined above.

The parameter `Directed` (`Directed := true` by default) may be used to return the Cayley multigraph of A as an undirected graph. When `Directed` is `false` and `Labelled` is `true`, any self-loop, which arises when a generator fixes a vertex, is omitted, since a labelled undirected self-loop is not supported.

By default, the generating set X used is `Generators(A)`. The optional parameter `GenSet` may be set to a set or sequence of elements of A to use as X instead; note that it is *not* checked that the given generating set actually generates A.

`SchreierMultiGraph(A, B : parameter): Grp, Grp -> Grph, GrphVertSet, GrphEdgeSet`

```
Labelled: BoolElt                    Default: true
Directed: BoolElt                    Default: true
GenSet  : SetEnum                    Default:
```

Given a finite group A defined on the generating set X and a subgroup B of A, construct the Schreier coset multigraph S for A over B, relative to X. The graph S is defined as follows: The vertices correspond to the cosets of B in A, and, for each x in X and each coset u of B in A, there is an edge from u to $u * x$. As with `CayleyMultiGraph`, when `Directed` is `false` and x is an involution, the unordered pair of cosets $\{u, u * x\}$ arises from both cosets, producing two parallel edges between them.

The graph is available in both a labelled and an unlabelled version and directed and undirected versions. These versions are controlled by the parameters `Labelled` and `Directed`, which are both `true` by default. As for `CayleyMultiGraph`, when `Directed` is `false` and `Labelled` is `true`, any self-loop, which arises when a generator stabilises a coset, is omitted, since a labelled undirected self-loop is not supported.

By default, the generating set X used is `Generators(A)`. The optional parameter `GenSet` may be set to a set or sequence of elements of A to use as X instead; note that it is *not* checked that the given generating set actually generates A.

Example: Cayley and Schreier Multigraphs of a Group (ex-7337d0)

The symmetric group of degree 3 is generated by a 3-cycle and an involution. Taking the Cayley multigraph over that generating set, each pair of elements joined by the involution is joined by two parallel edges, so the multigraph has three more edges than the Cayley graph on the same group.

```
> G := SymmetricGroup(3);
> C := CayleyMultiGraph(G : Directed := false, Labelled := false);
> C;
Multigraph
Vertex  Neighbours

1       4 3 3 2 ;
2       5 5 4 1 ;
3       5 1 6 1 ;
4       6 6 1 2 ;
5       6 2 3 2 ;
6       4 5 4 3 ;

> #EdgeSet(C);
12
> #EdgeSet(CayleyGraph(G : Directed := false));
9
```

The Schreier multigraph over the subgroup generated by the involution has one vertex for each of the three cosets. A generator that fixes a coset gives a self-loop, which the directed form keeps.

```
> S := SchreierMultiGraph(G, sub<G | (1,2)> : Labelled := false);
> S;
Multidigraph
Vertex  Neighbours

1       1 2 ;
2       3 3 ;
3       2 1 ;
```

163.5.3 Incremental Construction of Multigraphs

The complete functionality for adding and removing vertices or edges in simple graphs (see Subsection *Incremental Construction of Graphs*) is also available for multigraphs.

Note that some functions adding an edge to a multigraph also return the newly created edge. This feature is useful when there is a need to determine the index of this edge in the adjacency list of the

modified multigraph. Also, there is the possibility of removing *all* the edges from u to v for given vertices u and v of the multigraphs.

Further, whenever a vertex or an edge is added or removed from a graph, the existing vertex labels and the edge labels or capacities or weights are retained. The support of the graph is retained in all cases except when adding a new vertex.

Unless otherwise specified, each of the functions described in this section returns three values:

(i) The multigraph G;

(ii) The vertex-set V of G;

(iii) The edge-set E of G.

Adding Vertices

`G + n: GrphMult, RngIntElt -> GrphMult`

Given a non-negative integer n, adds n new vertices to the multigraph G. The existing vertex labels and edge labels or capacities or weights are retained, but the support will become the standard support.

`G +:= n: GrphMult, RngIntElt`

`AddVertex(~G): GrphMult`

`AddVertices(~G, n): GrphMult, RngIntElt`

The procedural version of the previous function. `AddVertex` adds one vertex only to G.

`AddVertex(~G, l): GrphMult, .`

Given a graph G and a label l, adds a new vertex with label l to G. The existing vertex labels and edge labels or capacities or weights are retained, but the support will become the standard support.

`AddVertices(~G, n, L): GrphMult, RngIntElt, SeqEnum`

Given a graph G and a non-negative integer n, and a sequence L of n labels, adds n new vertices to G with labels from L. The existing vertex labels and edge labels or capacities or weights are retained, but the support will become the standard support.

Removing Vertices

`G - v: GrphMult, GrphVert -> GrphMult`

`G - U: GrphMult, { GrphVert } -> GrphMult`

Given a vertex v of G, or a set U of vertices of G, removes v (or the vertices in U) from G. The support, vertex labels, and edge labels or capacities or weights are retained.

`G -:= v: GrphMult, GrphVert`

`G -:= U: GrphMult, { GrphVert }`

`RemoveVertex(~G, v): GrphMult, GrphVert`

`RemoveVertices(~G, U): GrphMult, { GrphVert }`

The procedural versions of the previous functions.

Adding Edges

`G + { u, v }: GrphMultUnd, {{ GrphVert, GrphVert } } -> GrphMultUnd, GrphEdge`

`G + [ u, v ]: GrphMultDir, { [ GrphVert, GrphVert ] } -> GrphMultDir, GrphEdge`

`G + [ u, v ]: GrphNet, { [ GrphVert, GrphVert ] } -> GrphNet, GrphEdge`

Given a graph G and a pair of vertices of G, add the edge to G described by this pairs. If G is undirected then the edge must be given as a set of (two) vertices, if G is directed the edge is given as a sequence of (two) vertices. If G is a network, then the edge is added with a capacity of 1 (0 if a loop). The support, vertex labels, and edge labels or capacities or weights are retained. This set of functions has two return values: The first is the modified graph and the second is the newly created edge. This feature is especially useful when adding parallel edges.

`G + { { u, v } }: GrphMultUnd, { { GrphVert, GrphVert } } -> GrphMultUnd`

`G + [ { u, v } ]: GrphMultUnd, [ { GrphVert, GrphVert } ] -> GrphMultUnd`

`G + { [ u, v ] }: GrphMultDir, { [ GrphVert, GrphVert ] } -> GrphDir`

`G + [ [ u, v ] ]: GrphMultDir, [ [ GrphVert, GrphVert ] ] -> GrphDir`

`G + { [ u, v ] }: GrphNet, { [ GrphVert, GrphVert ] } -> GrphNet`

`G + [ [ u, v ] ]: GrphNet, [ [ GrphVert, GrphVert ] ] -> GrphNet`

Given a graph G and a set or a sequence of pairs of vertices of G, add the edges to G described by these pairs. If G is undirected then the edges must be given as a set of (two) vertices, if G is directed the edges are given as a sequence of (two) vertices. If G is a network, then the edges are added with a capacity of 1 (0 if a loop). The support, vertex labels, and edge labels or capacities or weights are retained.

`G +:= { u, v }: GrphMultUnd, { GrphVert, GrphVert }`

`G +:= [ u, v ]: GrphDir, [ GrphVert, GrphVert ]`

`G +:= [ u, v ]: GrphNet, [ GrphVert, GrphVert ]`

`G +:= { { u, v } }: GrphMultUnd, { { GrphVert, GrphVert } }`

`G +:= [ { u, v } ]: GrphMultUnd, [ { GrphVert, GrphVert } ]`

`G +:= { [ u, v ] }: GrphMultDir, { [ GrphVert, GrphVert ] }`

`G +:= [ [ u, v ] ]: GrphMultDir, [ [ GrphVert, GrphVert ] ]`

`G +:= { [ u, v ] }: GrphNet, { [ GrphVert, GrphVert ] }`

`G +:= [ [ u, v ] ]: GrphNet, [ [ GrphVert, GrphVert ] ]`

The procedural versions of the previous four functions.

`AddEdge``(G, u, v): GrphMult, GrphVert, GrphVert -> GrphMult, GrphEdge`

Given a graph G, two vertices of G u and v, returns a new edge between u and v. If G is a network, the edge is added with a capacity of 1 (0 if loop). The support, vertex labels, and edge labels or capacities or weights are retained. This function has two return values: The first is the modified graph and the second is the newly created edge. This feature is especially useful when adding parallel edges.

`AddEdge``(G, u, v, l): GrphMultUnd, GrphVert, GrphVert, . -> GrphMult, GrphEdge`

`AddEdge``(G, u, v, l): GrphMultDir, GrphVert, GrphVert, . -> GrphMultDir, GrphEdge`

Given a graph G which is not a network, two vertices of G u and v, and a label l, adds a new edge with label l between u and v. The support, vertex labels, and edge labels or capacities or weights are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

`AddEdge``(G, u, v, c): GrphNet, GrphVert, RngIntElt, . -> GrphNet, GrphEdge`

Given a network G, two vertices of G u and v, and a non-negative integer c, adds a new edge from u to v with capacity c. The support, vertex labels, and edge labels or capacities or weights are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

`AddEdge``(G, u, v, c, l): GrphNet, GrphVert, GrphVert, RngIntElt, . -> GrphNet, GrphEdge`

Given a network G, two vertices of G u and v, a non-negative integer c, and a label l, adds a new edge from u to v with capacity c and label l. If G is a network, then add a new edge with label l and capacity c between u and v. The support, vertex labels, and edge labels or capacities or weights are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

`AddEdge``(~G, u, v): GrphMult, GrphVert, GrphVert`

`AddEdge``(~G, u, v, l): GrphMultUnd, GrphVert, GrphVert, .`

`AddEdge``(~G, u, v, l): GrphMultDir, GrphVert, GrphVert, .`

`AddEdge``(~G, u, v, c): GrphNet, GrphVert, GrphVert, RngIntElt`

`AddEdge(~G, u, v, c, l): GrphNet, GrphVert, GrphVert, RngIntElt, .`

Procedural versions of the previous functions for adding edges to a graph.

`AddEdges(G, S): GrphMultUnd, { { GrphVert, GrphVert } } -> GrphMultUnd`

`AddEdges(G, S): GrphMultUnd, [ { GrphVert, GrphVert } ] -> GrphMultUnd`

`AddEdges(G, S): GrphMultDir, { [ GrphVert, GrphVert ] } -> GrphMultDir`

`AddEdges(G, S): GrphMultDir, [ [ GrphVert, GrphVert ] ] -> GrphMultDir`

`AddEdges(G, S): GrphNet, { [ GrphVert, GrphVert ] } -> GrphNet`

`AddEdges(G, S): GrphNet, [ [ GrphVert, GrphVert ] ] -> GrphNet`

Given a graph G, and a sequence or set S of pairs of vertices of G, the edges specified in S are added to G. The elements of S must be sets or sequences of two vertices of G, depending upon whether G is undirected or directed respectively.

If G is a network, the edges are added with a capacity of 1 (0 if a loop). The support, vertex labels, and edge labels or capacities or weights are retained.

`AddEdges(G, S, L): GrphMult, SeqEnum, SeqEnum -> GrphMult`

Given a graph G, a sequence S of pairs of vertices of G, and a sequence L of labels of the same length, the edges specified in S are added to G with its corresponding label as given in L. The elements of S must be sets or sequences of two vertices of G, depending upon whether G is undirected or directed respectively. If G is a network, the edges are added with a capacity of 1 (0 if loop). The support, vertex labels, and edge labels or capacities or weights are retained.

`AddEdges(~G, S): GrphMultUnd, { { GrphVert, GrphVert } }`

`AddEdges(~G, S): GrphMultUnd, [ { GrphVert, GrphVert } ]`

`AddEdges(~G, S): GrphMultDir, { [ GrphVert, GrphVert ] }`

`AddEdges(~G, S): GrphMultDir, [ [ GrphVert, GrphVert ] ]`

`AddEdges(~G, S): GrphNet, { [ GrphVert, GrphVert ] }`

`AddEdges(~G, S): GrphNet, [ [ GrphVert, GrphVert ] ]`

`AddEdges(~G, S, L): GrphMult, SeqEnum, SeqEnum`

These are procedural versions of the previous functions for adding edges to a graph.

Removing Edges

`G - e: GrphMult, GrphEdge -> GrphMult`

`G - { e }: GrphMult, { GrphEdge } -> GrphMult`

Given an edge e or a set S of edges of a multigraph G, this function creates a graph that corresponds to G with the edge e (respectively, set of edges S) removed. The resulting multigraph will have vertex-set $V(G)$ and edge-set $E(G) \setminus \{e\}$ (respectively, $E(G) \setminus S$). The support, vertex labels, and edge labels are retained on the remaining edges.

`G - { { u, v } }: GrphMultUnd, { { GrphVert, GrphVert } } -> GrphMultUnd`

`G - { [u, v] }: GrphMultDir, { [ GrphVert, GrphVert ] } -> GrphMultDir`

Given a graph G and a set S of pairs $\{u, v\}$ or $[u, v]$, u, v vertices of G, this function forms the graph having vertex-set $V(G)$ and edge-set $E(G) - S$. That is, the graph returned is the same as G except that *all* the edges specified by pairs in S have been removed. An edge is represented as a set if G is undirected, and as a sequence otherwise. The support, vertex labels, and edge labels are retained on the remaining edges.

`G -:= e: GrphMult, GrphEdge`

`G -:= { e }: GrphMult, { GrphEdge }`

`G -:= { { u, v } }: GrphMultUnd, { { GrphVert, GrphVert rbrace }`

`G -:= { [u, v] }: GrphMultDir, { [ GrphVert, GrphVert ] }`

`RemoveEdge(~G, e): GrphMult, GrphEdge`

`RemoveEdges(~G, S): GrphMult, { GrphEdge }`

`RemoveEdge(~G, u, v): GrphMult, GrphVert, GrphVert`

`RemoveEdges(~G, S): GrphMultDir, { { GrphVert, GrphVert } }`

`RemoveEdges(~G, S): GrphMultDir, { [ GrphVert, GrphVert ] }`

These operations represent procedural versions of the previous functions. Whenever an edge is represented as a pair $\{u, v\}$ or $[u, v]$ of vertices of G, it is assumed that *all* the edges from u to v are to be removed from G.

163.5.4 Vertex Insertion, Contraction

As in the case of simple graphs (see Subsection *Constructing Complements, Line Graphs; Contraction, Switching*) it is possible to insert a vertex in a multigraph edge. The new edges thus created will be unlabelled with their capacities and weights set as follows: (These rules apply regardless as to whether the edge-set is capacitated and/or weighted).

Let e be an edge from u to v with capacity c and weight w in a multigraph G. After the insertion of a new vertex x in e, the edge e will be replaced by two edges, one from u to x and the other from x to v, *both* having capacity c and weight w.

These rules apply regardless as to whether the edge-set is capacitated and/or weighted.

The contraction operation can only be applied to a pair of vertices, since contracting a single multigraph edge which might have parallel edges is meaningless. The graph's support and vertex/edges decorations are retained when contracting its edges.

Each of the functions described below returns three values:

(i) The multigraph G;

(ii) The vertex-set V of G;

(iii) The edge-set E of G.

`InsertVertex(e): GrphEdge -> GrphMult`

Given an edge e of the multigraph G, this function inserts a new vertex of degree 2 in e. If appropriate, the two new edges that replace e will have the same capacity and weight as e. They will be unlabelled. The vertex labels and the edge decorations of G are retained, but the resulting graph will have standard support.

`InsertVertex(T): { GrphEdge} -> GrphMult`

Given a set T of edges belonging to the multigraph G, this function inserts a vertex of degree 2 in each edge belonging to the set T.

`Contract(e): GrphEdge -> GrphMult`

Given an edge $e = \{u, v\}$ of the graph G, form the graph obtained by removing the edge e and then identifying the vertices u and v. New parallel edges and new loops may result from this operation but any new loop will be assigned zero capacity. With the above exception, the edge decorations are retained, as are the support and the vertex labels of G.

`Contract(u, v): GrphVert, GrphVert -> GrphMult`

Given vertices u and v belonging to the multigraph G, this function returns the multigraph obtained by identifying the vertices u and v. New parallel edges and new loops may result from this operation but any new loop will be assigned zero capacity. With the above exception, the edge decorations are retained, as are the support and the vertex labels of G.

`Contract(S): { GrphVert} -> GrphMult`

Given a set S of vertices belonging to the multigraph G, this function returns the multigraph obtained by identifying all of the vertices in S.

163.5.5 Unions of Multigraphs

Of the union operations available for simple graphs (see Section *Unions and Products of Graphs*) only `Union` and `EdgeUnion` have been implemented for multigraphs. It is straightforward to write Magma code for other union functions with multigraphs.

In contrast with the other standard graph constructions, the support, vertex labels, and edge decorations are generally *not* handled by the functions listed below. Thus, the resulting graph will always have standard support and no vertex labels nor edge decorations. The one exception occurs in the case of networks, where edge capacities are properly handled.

`Union(G, H): GrphMultUnd, GrphMultUnd -> GrphMultUnd`

`Union(G, H): GrphMultDir, GrphMultDir -> GrphMultDir`

`G join H: GrphMultUnd, GrphMultUnd -> GrphMultUnd`

`G join H: GrphMultDir, GrphMultDir -> GrphMultDir`

Given multi(di)graphs G and H with disjoint vertex sets $V(G)$ and $V(H)$ respectively, this function constructs their union, i.e. the multi(di)graph with vertex-set $V(G) \cup V(H)$, and edge-set $E(G) \cup E(H)$. The resulting multi(di)graph has standard support and no vertex labels nor edge decorations.

`Union(N, H): GrphNet, GrphNet -> GrphNet`

`N join H: GrphNet, GrphNet -> GrphNet`

Given networks N and H with disjoint vertex sets $V(N)$ and (H) respectively, construct their union, i.e. the network with vertex-set $V(N) \cup V(H)$, and edge-set $E(N) \cup E(H)$. The resulting network has standard support and capacities but neither vertex labels, edge labels nor weights retained.

`& join S: [ MultiUnd ] -> GrphMultUnd`

`& join S: [ GrphMultDir ] -> GrphMultDir`

`& join S: [ GrphNet ] -> GrphNet`

`& join S: { MultiUnd } -> GrphMultUnd`

`& join S: { GrphMultDir } -> GrphMultDir`

`& join S: { GrphNet } -> GrphNet`

The union of the multigraphs or networks in the sequence or the set S.

`EdgeUnion(G, H): GrphMultUnd, GrphMultUnd -> GrphMultUnd`

`EdgeUnion(G, H): GrphMultDir, GrphMultDir -> GrphMultDir`

Given multi(di)graphs G and H having the same number of vertices, construct their edge union K. This construction identifies the i-th vertex of G with the i-th vertex of H for all i. The edge union has the same vertex-set as G (and hence as H) and there is an edge from u to v in K if and only if there is an edge from u to v in either G or H. The resulting multi(di)graph has standard support but neither vertex labels nor edge decorations.

`EdgeUnion(N, H): GrphNet, GrphNet -> GrphNet`

Given networks N and H having the same number of vertices, construct their edge union K. This construction identifies the i-th vertex of N with the i-th vertex of H for all i. The edge union has the same vertex-set as N (and hence as H) and there is an edge $[u, v]$ with capacity c in K if and only if either there is an edge $[u, v]$ with capacity c in N or if there is an edge $[u, v]$ with capacity c in H. The resulting network has standard support and the inherited edge capacities but neither vertex labels, edge labels nor weights.

163.6 Conversion Functions

Conversion functions do not preserve a graph's support and vertex/edge decorations. That is, the resulting graph has standard support and no vertex/edge decorations. A slight exception to this rule occurs when the resulting graph is a network (of type `GrphNet`) and is described in detail in *`UnderlyingNetwork`*.

163.6.1 Orientated Graphs

The rules followed in building an orientated graph from an undirected graph are the same as those described for simple graphs (see Section *Converting between Graphs and Digraphs*).

`OrientatedGraph(G): GrphMultUnd -> GrphMultDir`

Given a multigraph G, produce a multidigraph D whose vertex-set is the same as that of G and whose edge-set consists of the edges of G, each given a direction. The edges of D are always directed from the lower numbered vertex to the higher numbered vertex. Thus, if G contains the edge $\{u, v\}$, then D will have the edge $[u, v]$ if $u < v$, otherwise the edge $[u, v]$. If G has a loop at u, then D will have a directed loop at u.

163.6.2 Converse

`Converse(G): GrphMultDir -> GrphMultDir`

Given a multidigraph G with edge-set E, produce a multidigraph D whose vertex-set is the same as that of G and whose edge-set is $\{[u, v] : [v, u] \in E\}$.

163.6.3 Converting between Simple Graphs and Multigraphs

Any simple (di)graph can be converted into a multi(di)graph and any multi(di)graph can be converted into a simple (di)graph. The resulting graph has standard support and neither vertex labels nor edge decorations, unless it is a network, in which case all the edges in the resulting graph are assigned a capacity of 1 (0 if loops).

Let G be a graph and e an edge of G from u to v and let H be the graph resulting from the conversion. If G and H are both undirected or both directed then e is also an edge of H. If G is undirected while H is undirected then both edges $[u, v]$ and $[v, u]$ are edges of H. If G is directed while H is directed then the edge $\{u, v\}$ is an edge of H.

Since these conversion functions do not retain the original graph's support and vertex/edge decorations, they may also be used when requiring a copy of a graph G without G's support and vertex/edge decorations.

`UnderlyingGraph(G): GrphMult -> GrphUnd, GrphVertSet, GrphEdgeSet`

`UnderlyingGraph(G): Grph -> GrphUnd, GrphVertSet, GrphEdgeSet`

The underlying simple graph of the graph G. The support and vertex/edge decorations of G are not retained.

`UnderlyingDigraph(G): GrphMult -> GrphDir, GrphVertSet, GrphEdgeSet`

`UnderlyingDigraph(G): Grph -> GrphDir, GrphVertSet, GrphEdgeSet`

The underlying simple digraph of the graph G. The support and vertex/edge decorations of G are not retained.

`UnderlyingMultiGraph(G): Grph -> GrphMultUnd, GrphVertSet, GrphEdgeSet`

`UnderlyingMultiGraph(G): GrphMult -> GrphMultUnd, GrphVertSet, GrphEdgeSet`

The underlying multigraph of the graph G. The support and vertex/edge decorations of G are not retained.

`UnderlyingMultiDigraph(G): Grph -> GrphMultDir, GrphVertSet, GrphEdgeSet`

`UnderlyingMultiDigraph(G): GrphMult -> GrphMultDir, GrphVertSet, GrphEdgeSet`

The underlying multidigraph of the graph G. The support and vertex/edge decorations of G are not retained.

`UnderlyingNetwork(G): Grph -> GrphNet, GrphVertSet, GrphEdgeSet`

`UnderlyingNetwork(G): GrphMult -> GrphNet, GrphVertSet, GrphEdgeSet`

The underlying network of the graph G. The support and vertex/edge decorations of G are not retained except when G is a network in which case only the edge capacities are retained. If G is not a network, then all the edge capacities are set to 1 (0 for loops).

163.7 Elementary Invariants and Predicates for Multigraphs

Most but not all of the invariants and predicates that apply to simple graphs (see Sections *Elementary Invariants of a Graph* and *Elementary Graph Predicates*) also apply to multigraphs. We list them below.

Let G and H be two graphs. For clarity, we list here once again the conditions under which G is equal to H and H is a subgraph of G.

The graphs G and H are equal if and only if:

- they are of the same type,
- they are structurally identical,
- they have the same support,
- they have identical vertex and edge labels,
- if applicable, the total capacity from u to v in G is equal to the total capacity from u to v in H.

Also, H is a subgraph of G if and only if:

- they are of the same type,
- H is a structural subgraph of G,
- any vertex `v` in H has the same support as the vertex `VertexSet(G)!v` in G,
- any vertex `v` in H has the same label as the vertex `VertexSet(G)!v` in G,
- any edge `e` in H has the same label as the edge `EdgeSet(G)!e` in G,
- if applicable, the total capacity from u to v in G is at least as large as the total capacity from u to v in H.

Note that the truth value of the above two tests is not dependent on the weights of the edges of the graphs, should these edges be weighted.

Finally, we have introduced a few predicates to help users determine if a general graph is simple or not, undirected or not.

`Order(G): GrphMult -> RngIntElt`

`NumberOfVertices(G): GrphMult -> RngIntElt`

The number of vertices of the graph G.

`Size(G): GrphMult -> RngIntElt`

`NumberOfEdges(G): GrphMult -> RngIntElt`

The number of edges of the graph G.

`u adj v: GrphVert, GrphVert -> BoolElt`

Let u and v be two vertices of the same graph G. If G is undirected, returns `true` if and only if u and v are adjacent. If G is directed, returns `true` if and only if there is an edge directed from u to v.

`e adj f: GrphEdge, GrphEdge -> BoolElt`

Let e and f be two edges of the same graph G. If G is undirected, returns `true` if and only if e and f share a common vertex. If G is directed, returns `true` if and only if the terminal vertex of e (f) is the initial vertex of f (e).

`u notadj v: GrphVert, GrphVert -> BoolElt`

The negation of the `adj` predicate applied to vertices.

`e notadj f: GrphEdge, GrphEdge -> BoolElt`

The negation of the `adj` predicate applied to edges.

`u in e: GrphVert, GrphEdge -> BoolElt`

Let u be a vertex and e an edge of a graph G. Returns `true` if and only if u is an end-vertex of e.

`u notin e: GrphVert, GrphEdge -> BoolElt`

The negation of the `in` predicate applied to a vertex with respect to an edge.

`G eq H: GrphMultUnd, GrphMultUnd -> BoolElt`

`G eq H: GrphMultDir, GrphMultDir -> BoolElt`

`G eq H: GrphNet, GrphNet -> BoolElt`

Returns `true` if and only if the graphs G and H are equal, that is if and only if they are structurally equal and are compatible with respect to their support, vertex and edge labels, and edge capacities (see the introduction to this section).

`IsSubgraph(G, H): GrphMultUnd, GrphMultUnd -> BoolElt`

`IsSubgraph(G, H): GrphMultDir, GrphMultDir -> BoolElt`

`IsSubgraph(G, H): GrphNet, GrphNet -> BoolElt`

Returns `true` if and only if H is a subgraph of G, that is, if and only if H is a structural subgraph of G and the graphs are compatible with respect to their support, vertex and edge labels, and edge capacities (see the introduction to this section).

`IsBipartite(G): GrphMultUnd -> BoolElt`

Returns `true` if and only if the graph G is bipartite.

`Bipartition(G): GrphMultUnd -> [ { GrphVert} ]`

Given a bipartite graph G, return its two partite sets in the form of a pair of subsets of $V(G)$.

`IsRegular(G): GrphMult -> BoolElt`

Returns `true` if and only if G is a regular graph.

`IsComplete(G): GrphMult -> BoolElt`

Returns `true` if and only if the graph G, on n vertices, is the complete graph on n vertices.

`IsEmpty(G): GrphMult -> BoolElt`

Returns `true` if and only if the edge-set of the graph is empty.

`IsNull(G): GrphMult -> BoolElt`

Returns `true` if and only if the vertex-set of the graph is empty.

`IsSimple(G): GrphMult -> BoolElt`

`IsSimple(G): Grph -> BoolElt`

Returns `true` if and only if G is a simple graph.

`IsUndirected(G): GrphMult -> BoolElt`

`IsUndirected(G): Grph -> BoolElt`

Returns `true` if and only if G is a undirected graph.

`IsDirected(G): GrphMult -> BoolElt`

`IsDirected(G): Grph -> BoolElt`

Returns `true` if and only if G is a directed graph.

163.8 Adjacency and Degree

The adjacency and degree functionalities that apply to simple graphs (see *Adjacency and Degree*) similarly apply to multigraphs.

163.8.1 Adjacency and Degree Functions for Multigraphs

`Degree(u): GrphVert -> RngIntElt`

Given a vertex u of a graph G, return the degree of u, ie the number of edges incident to u.

`Alldeg(G, n): GrphMultUnd, RngIntElt -> { GrphVert}`

Given a multigraph G, and a non-negative integer n, return the set of all vertices of G that have degree equal to n.

`MaximumDegree(G): GrphMultUnd -> RngIntElt, GrphVert`

`Maxdeg(G): GrphMultUnd -> RngIntElt, GrphVert`

The maximum of the degrees of the vertices of the multigraph G. This function returns two values: the maximum degree, and a vertex of G having that degree.

`MinimumDegree(G): GrphMultUnd -> RngIntElt, GrphVert`

`Mindeg(G): GrphMultUnd -> RngIntElt, GrphVert`

The minimum of the degrees of the vertices of the multigraph G. This function returns two values: the minimum degree, and a vertex of G having that degree.

`DegreeSequence(G): GrphMultUnd -> [ { GrphVert} ]`

Given a multigraph G such that the maximum degree of any vertex of G is r, return a sequence D of length $r+1$, such that $D[i]$, $1 \leq i \leq r+1$, is the number of vertices in G having degree $i-1$.

`Neighbours(u): GrphVert -> { GrphVert}`

`Neighbors(u): GrphVert -> { GrphVert}`

Given a vertex u of a graph G, return the set of vertices of G that are adjacent to u.

`IncidentEdges(u): GrphVert -> { GrphEdge}`

Given a vertex u of a graph G, return the set of all edges incident with the vertex u.

163.8.2 Adjacency and Degree Functions for Multidigraphs

`InDegree(u): GrphVert -> RngIntElt`

The number of edges directed into the vertex u belonging to a multidigraph.

`OutDegree(u): GrphVert -> RngIntElt`

The number of edges of the form $[u, v]$ where u is a vertex belonging to a multidigraph.

`MaximumInDegree(G): GrphMultDir -> RngIntElt, GrphVert`

`Maxindeg(G): GrphMultDir -> RngIntElt, GrphVert`

The maximum indegree of the vertices of the multidigraph G. This function returns two values: the maximum indegree, and the first vertex of G having that degree.

`MinimumInDegree(G): GrphMultDir -> RngIntElt, GrphVert`

`Minindeg(G)): GrphMultDir -> RngIntElt, GrphVert`

The minimum indegree of the vertices of the multidigraph G. This function returns two values: the minimum indegree, and the first vertex of G having that degree.

`MaximumOutDegree(G): GrphMultDir -> RngIntElt, GrphVert`

`Maxoutdeg(G): GrphMultDir -> RngIntElt, GrphVert`

The maximum outdegree of the vertices of the multidigraph G. This function returns two values: the maximum outdegree, and the first vertex of G having that degree.

`MinimumOutDegree(G): GrphMultDir -> RngIntElt, GrphVert`

`Minoutdeg(G): GrphMultDir -> RngIntElt, GrphVert`

The minimum outdegree of the vertices of the multidigraph G. This function returns two values: the minimum outdegree, and the first vertex of G having that degree.

`Degree(u): GrphVert -> RngIntElt`

Given a vertex u belonging to the multidigraph G, return the total degree of u, i.e. the sum of the in–degree and out–degree for u.

`MaximumDegree(G): GrphMultDir -> RngIntElt, GrphVert`

`Maxdeg(G): GrphMultDir -> RngIntElt, GrphVert`

The maximum total degree of the vertices of the multidigraph G. This function returns two values: the maximum total degree, and the first vertex of G having that degree.

`MinimumDegree(G): GrphMultDir -> RngIntElt, GrphVert`

`Mindeg(G): GrphMultDir -> RngIntElt, GrphVert`

The minimum total degree of the vertices of the multidigraph G. This function returns two values: the minimum total degree, and the first vertex of G having that degree.

`Alldeg(G, n): GrphMultDir, RngIntElt -> { GrphVert}`

Given a multidigraph G, and a non–negative integer n, return the set of all vertices of G that have total degree equal to n.

`DegreeSequence(G): GrphMultDir -> [ { GrphVert } ]`

Given a multidigraph G such that the maximum degree of any vertex of G is r, return a sequence D of length $r+1$, such that $D[i]$, $1 \leq i \leq r+1$, is the number of vertices in G having degree $i-1$.

`InNeighbours(u): GrphVert -> { GrphVert}`

`InNeighbors(u): GrphVert -> { GrphVert}`

Given a vertex u of a multidigraph G, return the set containing all vertices v such that $[v, u]$ is an edge in G, i.e. the initial vertex of all edges that are directed into the vertex u.

`OutNeighbours(u): GrphVert -> { GrphVert}`

`OutNeighbors(u): GrphVert -> { GrphVert}`

Given a vertex u of the multidigraph G, return the set of vertices v of G such that $[u, v]$ is an edge in G, i.e. the set of vertices v that are terminal vertices of edges directed from u to v.

`IncidentEdges(u): GrphVert -> { GrphEdge}`

Given a vertex u of a graph G, return the set of all edges incident with the vertex u, that is, the set of all edges incident into u and incident from u.

163.9 Connectedness

All the functions relating to connectivity issues are the same for simple graphs and multigraphs. See Section *Connectedness in a Graph* and its subsections for specific details about the algorithms underlying these functions.

163.9.1 Connectedness in a Multigraph

`IsConnected(G): GrphMultUnd -> BoolElt`

Returns `true` if and only if the undirected graph G is a connected graph.

`Components(G): GrphMultUnd -> [ { GrphVert } ]`

The connected components of the undirected graph G. These are returned in the form of a sequence of subsets of the vertex-set of G.

`Component(u): GrphVert -> GrphMult`

The subgraph corresponding to the connected component of the multigraph G containing vertex u. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

`IsSeparable(G): GrphMultUnd -> BoolElt`

Returns `true` if and only if the graph G is connected and has at least one cut vertex.

`IsBiconnected(G): GrphMultUnd -> BoolElt`

Returns `true` if and only if the graph G is biconnected. The graph G must be undirected.

`CutVertices(G): GrphMultUnd -> { GrphVert}`

The set of cut vertices for the connected undirected graph G (as a set of vertices).

`Bicomponents(G): GrphMultUnd -> [ { GrphVert } ]`

The biconnected components of the undirected graph G. These are returned in the form of a sequence of subsets of the vertex-set of G. The graph may be disconnected.

163.9.2 Connectedness in a Multidigraph

`IsStronglyConnected(G): GrphMultDir -> BoolElt`

Returns `true` if and only if the multidigraph G is strongly connected.

`IsWeaklyConnected(G): GrphMultDir -> BoolElt`

Returns `true` if and only if the multidigraph G is weakly connected.

`StronglyConnectedComponents(G): GrphMultDir -> [ { GrphVert} ]`

The strongly connected components of the multidigraph G. These are returned in the form of a sequence of subsets of the vertex-set of G.

`Component(u): GrphVert -> GrphMult`

The subgraph corresponding to the connected component of the multidigraph G containing vertex u. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

163.9.3 Triconnectivity for Multigraphs

We refer the reader to Subsection *Graph Triconnectivity* of the graphs chapter for details about the triconnectivity algorithm implemented here, and especially about the meaning of the `Splitcomponents` function. The triconnectivity algorithm applies to undirected graphs only.

`IsTriconnected(G): GrphMultUnd -> BoolElt`

Returns `true` if and only if G is triconnected. The graph G must be undirected.

`Splitcomponents(G): GrphMultUnd -> [ { GrphVert} ], [ [ GrphVert ]]`

The split components of the undirected graph G. These are returned in the form of a sequence of subsets of the vertex-set of G. The graph may be disconnected. The second return value returns the cut vertices and the separation pairs as sequences of one or two vertices respectively.

`SeparationVertices(G): GrphMultUnd -> [ [ GrphVert ]], [ { GrphVert} ]`

The cut vertices and/or the separation pairs of the undirected graph G as a sequence of sequences of one and/or two vertices respectively. The graph may be disconnected. The second return value returns the split components of G .

163.9.4 Maximum Matching in Bipartite Multigraphs

We refer the reader to Subsection *Maximum Matching in Bipartite Graphs* for information about the maximum matching algorithm.

`MaximumMatching(G : parameters): GrphMultUnd -> [ { GrphEdge rbrace ]`

```
Al: MonStg                    Default: "PushRelabel"
```

A maximum matching in the bipartite graph G. The matching is returned as a sequence of edges of G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

163.9.5 General Vertex and Edge Connectivity in Multigraphs and Multidigraphs

See Subsection *General Vertex and Edge Connectivity in Graphs and Digraphs* for details about the algorithms underlying the functions listed below. These functions apply to both undirected and directed graphs.

`VertexSeparator(G : parameters): GrphMult -> [ GrphVert ]`

`Al: MonStg`	`Default: "PushRelabel"`

If G is an undirected graph, a *vertex separator* for G is a smallest set of vertices S such that for any u, $v \in V(G)$, every path connecting u and v passes through at least one vertex of S.

If G is a directed graph, a vertex separator for G is a smallest set of vertices S such that for any u, $v \in V(G)$, every directed path from u to v passes through at least one vertex of S.

`VertexSeparator` returns the vertex separator of G as a sequence of vertices of G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`VertexConnectivity(G : parameters): GrphMult -> RngIntElt, [ GrphVert ]`

`Al: MonStg`	`Default: "PushRelabel"`

Returns the vertex connectivity of the graph G, the size of a minimum vertex separator of G. Also returns a vertex separator for G as the second value. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`IsKVertexConnected(G, k : parameters): GrphMult, RngIntElt -> BoolElt`

`Al: MonStg`	`Default: "PushRelabel"`

Returns `true` if the vertex connectivity of the graph G is at least k, `false` otherwise. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`EdgeSeparator(G : parameters): GrphMult -> [ GrphEdge ]`

`Al: MonStg`	`Default: "PushRelabel"`

If G is an undirected graph, an *edge separator* for G is a smallest set of edges T such that for any u, $v \in V(G)$, every path connecting u and v passes through at least one edge of T.

If G is a directed graph, an edge separator for G is a smallest set of edges T such that for any u, $v \in V(G)$, every directed path from u to v passes through at least one edge of T.

`EdgeSeparator` returns the edge separator of G as a sequence of edges of G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`EdgeConnectivity(G : parameters): GrphMult -> RngIntElt, [ GrphEdge ]`

`Al: MonStg`	`Default: "PushRelabel"`

Returns the edge connectivity of the graph G, the size of a minimum edge separator of G. Also returns as the second value an edge separator for G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`IsKEdgeConnected(G, k : parameters): GrphMult, RngIntElt -> BoolElt`

`Al: MonStg`	`Default: "PushRelabel"`

Returns `true` if the edge connectivity of the graph G is at least k, `false` otherwise. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

Example: Connectivity of a Multigraph `(ex-e96180)`

We demonstrate that the connectivity functions deal properly with multiple edges.

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2, 3} >, < 2, {2, 3} > >;
> G;
Multigraph
Vertex  Neighbours

1       3 2 3 2 ;
2       3 2 2 1 1 ;
3       2 1 1 ;

> EdgeConnectivity(G);
3
> EdgeSeparator(G);
[ < {3, 2}, 11 >, < {1, 2}, 5 >, < {1, 2}, 1 > ]
```

163.10 Spanning Trees

All the trees returned by the functions described below are returned as structural subgraphs of the original graph whose support and vertex and edge decorations are not retained in the resulting tree.

SpanningTree(G): GrphMultUnd -> GrphMultUnd, GrphVertSet, GrphEdgeSet

Given a connected undirected graph G, construct a spanning tree for G rooted at an arbitrary vertex of G. The spanning tree is returned as a structural subgraph of G, without the support, vertex, or edge decorations of G.

SpanningForest(G): GrphMult -> GrphMult, GrphVertSet, GrphEdgeSet

Given a graph G, construct a spanning forest for G. The forest is returned as a structural subgraph of G, without the support, vertex, or edge decorations of G.

BreadthFirstSearchTree(u): GrphVert -> GrphMult, GrphVertSet, GrphEdgeSet

BFSTree(u): GrphVert -> GrphMult

Given a vertex u belonging to the graph G, return a breadth-first search for G rooted at the vertex u. The tree is returned as a structural subgraph of G, without the support, vertex, or edge decorations of G. Note that G may be disconnected.

DepthFirstSearchTree(u): GrphVert -> GrphMult, GrphVertSet, GrphEdgeSet, SeqEnum

DFSTree(u): GrphVert -> GrphMult, GrphVertSet, GrphEdgeSet, SeqEnum

Given a vertex u belonging to the graph G, return a depth-first search tree T for G rooted at the vertex u. The tree T is returned as a structural subgraph of G, without the support, vertex, or edge decorations of G. Note that G may be disconnected.

The fourth return argument returns, for each vertex u of G, the tree order of u, that is, the order in which the vertex u has been visited while performing the depth-first search. If T does not span G then the vertices of G not in T are given tree order from `Order(T) + 1` to `Order(G)`.

163.11 Planar Graphs

The planarity algorithm implemented in Magma tests whether an undirected graph or multigraph is planar. If the graph is planar, then an embedding of the graph is produced, otherwise a Kuratowski subgraph is identified. For a thorough discussion of this algorithm, its implementation, and complexity, the reader is referred to Section *Planar Graphs*.

IsPlanar(G): GrphMultUnd -> BoolElt, GrphMultUnd

Tests whether the (undirected) graph G is planar. The graph may be disconnected. If the graph is non-planar then a Kuratowski subgraph of G is returned: That is, a subgraph of G homeomorphic to K_5 or $K_{3,3}$. The support and vertex/edge decorations of G are *not* retained in this (structural) subgraph.

`Obstruction(G): GrphMultUnd -> GrphMultUnd`

Returns a Kuratoswki obstruction if the graph is non-planar, or the empty graph if the graph is planar. The Kuratoswki graph is returned as a (structural) subgraph of G; the support and vertex/edge decorations are not retained.

`IsHomeomorphic(G: parameters): GrphMultUnd -> BoolElt`

Graph: MonStg	Default:

Tests if a graph is homeomorphic to either K_5 or $K_{3,3}$. The parameter `Graph` must be set to either "K5" or "K33"; it has no default setting.

`Faces(G): GrphMultUnd -> SeqEnum[GrphVert]`

Returns the faces of the planar graph G as sequences of the edges bordering the faces of G. If G is disconnected, then the face defined by an isolated vertex v is given as $[v]$.

`Face(u, v): GrphVert, GrphVert -> SeqEnum`

Returns the face of the planar graph G bordered by the directed edge $[u, v]$ as an ordered list of edges of G.

Note that a directed edge and an orientation determine a face uniquely: We can assume without loss of generality that the plane is given a clockwise orientation. Then given a directed edge $e = [u_1, v_1]$, the face defined by e is the ordered set of edges $[u_1, v_1], [u_2, v_2], \ldots, [u_m, v_m]$ such that $v_i = u_{i+1}$ for all i, $1 \leq i < m$, $v_m = u_1$, and for each $v_i = u_{i+1}$, the neighbours of v_i, u_i and v_{i+1}, are *consecutive* vertices in v_i's adjacency list whose order is *anti-clockwise*.

`Face(e): GrphEdge -> SeqEnum`

Let e be the edge u, v of the planar graph G (recall that G is undirected). Then `Face`(u, v) returns the face bordered by the directed edge $[u, v]$ as a sequence of edges of G.

`NFaces(G): GrphMultUnd -> RngIntElt`

`NumberOfFaces(G): GrphMultUnd -> RngIntElt`

Returns the number of faces of the planar graph G. In the case of a disconnected graph, an isolated vertex counts for one face.

`Embedding(G): GrphMultUnd -> SeqEnum`

Returns the planar embedding of the graph G as a sequence S where $S[i]$ is a sequence of edges incident from vertex i.

`Embedding(v): GrphVert -> SeqEnum`

Returns the ordered list of edges (in clockwise order say) incident from vertex v.

Example: Planar Embedding of a Multigraph (ex-c07270)

The purpose of the example is to show the embedding and faces of a graph with multiples edges and loops.

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multigraph
Vertex  Neighbours

1       2 3 2 ;
2       3 2 2 1 1 ;
3       2 1 ;

> IsPlanar(G);
true
> Faces(G);
[
```

(continues on next page)

(continued from previous page)

```
  [ < {1, 2}, 5 >, < {2, 1}, 1 > ],
  [ < {1, 2}, 1 >, < {2, 2}, 7 >, < {2, 3}, 9 >, < {3, 1}, 3 > ],
  [ < {1, 3}, 3 >, < {3, 2}, 9 >, < {2, 1}, 5 > ],
  [ < {2, 2}, 7 > ]
]
> Embedding(G);
[
  [ < {1, 2}, 5 >, < {1, 2}, 1 >, < {1, 3}, 3 > ],
  [ < {2, 3}, 9 >, < {2, 2}, 7 >, < {2, 1}, 1 >, < {2, 1}, 5 > ],
  [ < {3, 1}, 3 >, < {3, 2}, 9 > ]
]
```

Example: Planar Dual and Minimal Cuts (ex-d8a2c8)

We show how to construct the dual graph D of a planar graph G and how to find all its minimal cuts. The vertex set of the dual is the set of faces F of G where face f_i is adjacent to face f_2 if and only if f_1 and f_2 share a common edge in G. For the purpose of this example a cut of a graph G is defined as a set of edges which disconnects G.

Let us construct a small planar graph G and its dual D. For clarity, the support of D will be the standard support (we could have chosen it to be the set of faces of G).

```
> G := MultiGraph< 4 | {1, 2}, {1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 4}>;
> IsPlanar(G);
true
> Faces(G);
[
  [ < {1, 3}, 5 >, < {3, 2}, 7 >, < {2, 1}, 3 > ],
  [ < {1, 2}, 3 >, < {2, 1}, 1 > ],
  [ < {1, 2}, 1 >, < {2, 4}, 9 >, < {4, 3}, 11 >, < {3, 1}, 5 > ],
  [ < {3, 4}, 11 >, < {4, 2}, 9 >, < {2, 3}, 7 > ]
]
> F := {@ SequenceToSet(f) : f in Faces(G) @};
> D := MultiGraph< #F | >;
> mapG2D := [ 0 : i in [1..Max(Indices(G))] ];
> mapD2G := [ 0 : i in [1..Max(Indices(G))] ];
> for u in VertexSet(D) do
>     for v in VertexSet(D) do
>        if Index(v) le Index(u) then
>           continue;
>        end if;
>        M := F[ Index(u) ] meet F[ Index(v) ];
>        for e in M do
>           D, edge :=
>           AddEdge(D, VertexSet(D)!u, VertexSet(D)!v);
>
>           mapG2D[Index(e)] := Index(edge);
>           mapD2G[Index(edge)] := Index(e);
>        end for;
>     end for;
> end for;
>
> e_star := map< EdgeSet(G) -> EdgeSet(D) |
> x :-> EdgeSet(D).mapG2D[Index(x)],
> y :-> EdgeSet(G).mapD2G[Index(y)] >;
```

The map `e-star` is the bijection from G's edge-set into D's edge-set:

```
> for e in EdgeSet(G) do
>     e, "   ", e @ e_star;
> end for;
< {1, 3}, 5 >     < {1, 3}, 3 >
< {1, 2}, 3 >     < {1, 2}, 1 >
< {1, 2}, 1 >     < {2, 3}, 7 >
< {2, 4}, 9 >     < {3, 4}, 11 >
< {2, 3}, 7 >     < {1, 4}, 5 >
< {3, 4}, 11 >     < {3, 4}, 9 >
>
> for e in EdgeSet(D) do
>     e, "   ", e @@ e_star;
> end for;
< {1, 4}, 5 >     < {2, 3}, 7 >
< {1, 3}, 3 >     < {1, 3}, 5 >
< {1, 2}, 1 >     < {1, 2}, 3 >
< {2, 3}, 7 >     < {1, 2}, 1 >
< {3, 4}, 11 >     < {2, 4}, 9 >
< {3, 4}, 9 >     < {3, 4}, 11 >
```

If G is biconnected, then any of its faces is bounded by a cycle. From Euler's formula giving the number of faces in a graph, we deduce that the boundaries of the internal faces of G, which form a chordless cycle, form a basis for the cycle space of G.

It is a well-known fact that, if G is connected and planar, a set of edges E is the set of edges of a cycle in G if and only if $E^* = \{e^* : e \in E\}$ is a minimal cut in D. For more details, see [Die00, \$ 4].

From this we conclude that we can compute the minimal cuts generating the cut space of the dual graph D. We verify that G is biconnected, we compute the cut corresponding to each face of G, and verify that it is a minimal cut. All the cuts together form a generating set of the cut space of D. Had we not included the cut corresponding to the external face of G, we would have a basis of the cut space.

```
> IsBiconnected(G);
true
> for f in F do
>     Cut := { e @ e_star : e in f };
>     H := D;
>     RemoveEdges(~H, Cut);
>     assert not IsConnected(H);
>
>     for e in Cut do
>        C := Exclude(Cut, e);
>        H := D;
>        RemoveEdges(~H, C);
>        assert IsConnected(H);
>     end for;
> end for;
```

163.12 Distances, Shortest Paths, and Minimum Weight Trees

Two standard algorithms have been implemented for finding single-source shortest paths in weighted graphs. One is Dijsktra's algorithm for graphs without negative weight cycles, the second is Bellman-Ford's for those graphs with negative weight cycles. Dijsktra's algorithm is implemented either by a priority queue (binary heap) or a Fibonacci heap. The Fibonacci heap is asymptotically faster for sparse graphs. But for most practical purposes (graphs of small order) the binary heap outperforms the Fibonacci heap and is therefore chosen as the default data structure wherever relevant.

Johnson's algorithm has been chosen for the all-pairs shortest paths computation. Indeed, it outperforms the simpler Floyd's algorithm, especially as the graphs get larger.

Finally, Prim's algorithm is used to implement the minimum weight tree computation for undirected graphs. (The tree is a spanning tree if and only if the graph is connected.)

All the functions described below apply to general graphs whose edges are assigned a weight. If the graph under consideration is not weighted, then all its edges are assumed to have weight one. To assign weights to the edges of a graph, see Subsection *Edge Decorations*. Note that all the functions described below accept negatively weighted edges.

`Reachable(u, v : parameters): GrphVert, GrphVert -> BoolElt, RngElt`

`UseFibonacciHeap: Bool`	`Default: false`

Return `true` if and only there is a path from vertex u to vertex v. If true, also returns the distance between u and v.

`Distance(u, v : parameters): GrphVert, GrphVert -> RngElt`

`UseFibonacciHeap: Bool`	`Default: false`

Given vertices u and v in a graph G, computes the distance from u to v. Results in an error if there is no path in G from u to v.

`Distances(u : parameters): GrphVert -> Eseq`

`UseFibonacciHeap: Bool`	`Default: false`

Given a vertex u in a graph G, computes the sequence D of distances from u to v, v any vertex in G. Given any vertex v in G, let i be `Index(v)`. Then $D[i]$, if defined, is the distance from u to v. If there is no path from u to v, then the sequence element $D[i]$ is undefined.

`PathExists(u, v : parameters): GrphVert, GrphVert -> BoolElt, Eseq`

`UseFibonacciHeap: Bool`	`Default: false`

Return `true` if and only there is a path from vertex u to vertex v in the parent graph G. If so, also returns a shortest path from u to v as a sequence of edges of G.

`Path(u, v : parameters): GrphVert, GrphVert -> Eseq`
`ShortestPath(u, v : parameters): GrphVert, GrphVert -> Eseq`

`UseFibonacciHeap: Bool`	`Default: false`

Given vertices u and v in a graph G, computes a shortest path from u to v as a sequence of edges of G. Results in an error if there is no path in G from u to v.

`Paths(u : parameters): GrphVert -> Eseq`
`ShortestPaths(u : parameters): GrphVert -> Eseq`

`UseFibonacciHeap: Bool`	`Default: false`

Given a vertex u in a graph G, computes the sequence P of shortest paths from u to v, for every vertex v in G. Given any vertex v in G, let i be `Index(v)`. If there exists a path from u to v then $P[i]$ is a sequence of edges giving a shortest path from u to v. If there is no path from u to v, then the sequence element $P[i]$ is undefined.

`GeodesicExists(u, v : parameters): GrphVert, GrphVert -> BoolElt, Eseq`

`UseFibonacciHeap: Bool`	`Default: false`

Return `true` if and only there is a path from vertex u to vertex v in the parent graph G. If true, also returns a shortest path from u to v as a sequence of vertices of G.

`Geodesic(u, v : parameters): GrphVert, GrphVert -> Eseq`

```
UseFibonacciHeap: Bool                    Default: false
```

Given vertices u and v in a graph G, this function computes a shortest path from u to v as a sequence of vertices of G. An error results if there is no path in G from u to v.

`Geodesics(u : parameters): GrphVert -> Eseq`

```
UseFibonacciHeap: Bool                    Default: false
```

Given a vertex u in a graph G, this function computes the sequence P of shortest paths from u to v, for every vertex v in G. Given any vertex v in G, let i be `Index(v)`. If there exists a path from u to v then $P[i]$ is a sequence of vertices specifying a shortest path from u to v. If there is no path from u to v, then the sequence element $P[i]$ is undefined.

`HasNegativeWeightCycle(u : parameters): GrphVert -> BoolElt`

Return `true` if and only if there is a negative-weight cycle reachable from vertex u.

`HasNegativeWeightCycle(G): Grph -> BoolElt`

`HasNegativeWeightCycle(G): GrphMult -> BoolElt`

Return `true` if and only if the graph G has negative-weight cycles.

`AllPairsShortestPaths(G : parameters): Grph -> SeqEnum, SeqEnum`

`AllPairsShortestPaths(G : parameters): GrphMult -> SeqEnum, SeqEnum`

```
UseFibonacciHeap: Bool                    Default: false
```

Computes the all-pairs shortest paths. Let u and v be two vertices of a graph G and let $i = Index(u)$ and $j = Index(v)$. Let S_1 and S_2 be the two sequences returned by `AllPairsShortestPaths`, and let $s_1 = S_1[i]$ and $s_2 = S_2[i]$. Then $s_1[j]$, if defined, gives the distance from u to v and $s_2[j]$, if defined, gives the vertex preceding v in the shortest path from u to v. An error results if the graph G has a negative-weight cycle.

`MinimumWeightTree(u : parameters): GrphVert -> SeqEnum`

```
UseFibonacciHeap: Bool                    Default: false
```

Returns a minimum weight tree rooted at vertex u, u any vertex of an *undirected* graph G, as a subgraph of G. The tree spans G if and only if G is connected. The support of G as well as the vertex and edge decorations in G *are* transferred to the tree.

Example: Shortest Paths in a Multidigraph (ex-549372)

We create a weighted multidigraph.

```
> G := MultiDigraph< 5 | [1, 2], [1, 2], [1, 3], [2, 4], [3, 5], [3, 4], [4, 5] >
↪;
> E := EdgeSet(G);
> AssignWeight(~G, E.1, 1);
> AssignWeight(~G, E.2, 5);
> AssignWeight(~G, E.3, 10);
> AssignWeight(~G, E.4, 1);
> AssignWeight(~G, E.5, -5);
> AssignWeight(~G, E.6, 1);
> AssignWeight(~G, E.7, 2);
>
```

(continues on next page)

(continued from previous page)

```
>
> V := VertexSet(G);
> E := EdgeSet(G);
> for e in E do
>     e, "  ", Weight(e);
> end for;
< [1, 3], 3 >    10
< [1, 2], 2 >    5
< [1, 2], 1 >    1
< [2, 4], 4 >    1
< [3, 4], 6 >    1
< [3, 5], 5 >    -5
< [4, 5], 7 >    2
```

We verify that it has no negative weight cycle reachable from vertex 1, and that there is a path from vertex 1 to vertex 5.

```
> HasNegativeWeightCycle(V!1);
false
> b, d := Reachable(V!1, V!5);
> assert b;
> P := Path(V!1, V!5);
> G := Geodesic(V!1, V!5);
>
> d;
4
> P;
[ < [1, 2], 1 >, < [2, 4], 4 >, < [4, 5], 7 > ]
> G;
[ 1, 2, 4, 5 ]
```

Finally, we verify that the shortest path found has length 4.

```
> dP := 0;
> for e in P do
>     dP +:= Weight(e);
> end for;
> assert dP eq d;
```

Note that had we taken instead an undirected graph, we would have had to assign only positive weights to the edges of the graph: any undirected edge $\{u, v\}$ with negative weight results in the negative weight cycles $\{u, u\}$ and $\{v, v\}$.

Example: Minimum Weight Tree of a Multigraph (ex-8c3a20)

We create a weighted multigraph with one edge assigned a negative weight.

```
> G := MultiGraph< 5 | {1, 2}, {1, 2}, {1, 3}, {2, 4}, {3, 5}, {3, 4}, {4, 5} >;
> E := EdgeSet(G);
> AssignWeight(~G, E.1, 1);
> AssignWeight(~G, E.3, 5);
> AssignWeight(~G, E.5, 10);
> AssignWeight(~G, E.7, 1);
> AssignWeight(~G, E.9, -5);
> AssignWeight(~G, E.11, 1);
> AssignWeight(~G, E.13, 2);
>
```

(continues on next page)

(continued from previous page)

```
> V := VertexSet(G);
> E := EdgeSet(G);
> for e in E do
>     e, "  ", Weight(e);
> end for;
< {1, 3}, 5 >    10
< {1, 2}, 3 >    5
< {1, 2}, 1 >    1
< {2, 4}, 7 >    1
< {3, 4}, 11 >    1
< {3, 5}, 9 >    -5
< {4, 5}, 13 >    2
```

We compute a minimum weight spanning tree rooted at vertex 1.

```
> T := MinimumWeightTree(V!1);
> ET := EdgeSet(T);
> for e in ET do
>     e, "  ", Weight(e);
> end for;
< {1, 2}, 1 >    1
< {2, 4}, 5 >    1
< {3, 5}, 7 >    -5
< {3, 4}, 3 >    1
```

We compute any other spanning tree rooted at vertex 1 (say a depth first tree using *DFSTree*), and verify that the weights of the edges in G corresponding to the edges of the tree sum up to a total weight which is no smaller than the weight of the minimum weight spanning tree.

```
> DFST := DFSTree(V!1);
> EDT := EdgeSet(DFST);
> for e in EDT do
>     u := InitialVertex(e);
>     v := TerminalVertex(e);
>     w := Min([ Weight(edge) : edge in Edges(V!u, V!v) ]);
>     e, "  ", w;
> end for;
< {1, 3}, 1 >    10
< {2, 4}, 7 >    1
< {3, 4}, 3 >    1
< {4, 5}, 5 >    2
```

163.13 Bibliography

- [Die00] Reinhard Diestel. *Graph Theory, Second Edition.* Springer, 2000.

Chapter 164

Networks

164.1 Introduction

Networks are an essential tool in modelling communication systems and dependence problems. A network is generally defined as a directed graph whose arcs are associated with a cost and a capacity; it may have multiple (i.e., parallel) edges.

The fundamental network flow problem is the minimum cost flow problem, that is, determining a maximum flow at minimum cost from a specified source to a specified sink. Specializations of this problem are the shortest path problem (where there is no capacity constraint) and the maximum flow problem (where there is no cost constraint). Some of the related problems are the minimum spanning tree problem (finding a spanning tree whose sum of the costs of its arcs is minimum), the matching problem (a pairing of the edges of the graph according to some criteria), and the multicommodity flow problem (where arcs may carry several flows of different nature). For a comprehensive monograph on networks, their implementation, and applications, see [AMO93].

For convenience we provide users with the Magma network object with type `GrphNet`. It differs from the Magma multidigraph object having type `GrphMultDir` in one respect only: its edges are always assumed to be capacitated. The edges of a network have a capacity of one by default, unless they are specifically assigned a capacity. The loops in a network always have capacity zero. Since networks are a specialisation of multidigraphs, all the functions applying to multidigraphs also apply to networks. Below we outline those few functions that specifically concern networks, namely, their construction.

164.2 Construction of Networks

Networks are constructed in a similar way to multidigraphs (Subsection *Construction of a General Multidigraph*). In this implementation the order n of a network is bounded by 134217722. See Section *Bounds on the Graph Order* for more details on this.

Let N be the network to be constructed. In all cases, whenever an edge $[u, v]$, $u \neq v$, is to be added to N, its capacity will be set to 1 (0 if a loop) unless either its capacity is explicitly given at construction time, or it is the edge of a network, in which case the capacity of the edge remains as it was in the original network.

As an example, if D is a digraph, then the edges of the network N constructed as `N := Network< Order(D) | D >`; will be all the edges of D whose capacity is set as 1 (or 0 if they are loops).

`Network<n | edges >: RngIntElt, List -> GrphNet, GrphVertSet, GrphEdgeSet`

`Network<S | edges >: SetEnum, List -> GrphNet, GrphVertSet, GrphEdgeSet`

`Network<S | edges >: SetIndx, List -> GrphNet, GrphVertSet, GrphEdgeSet`

Construct the network N with vertex-set $V = \{@v_1, v_2, \ldots, v_n@\}$ (where $v_i = i$ for each i if the first form of the constructor is used, or the ith element of the enumerated or indexed set S otherwise), and edge-set $E = \{e_1, e_2, \ldots, e_q\}$. This function returns three values: The network N, the vertex-set V of N; and the edge-set E of N.

The elements of E are specified by the list *edges*, where the items of *edges* may be objects of the following types:

(a) A pair $[v_i, v_j]$ of vertices in V. The directed edge $[v_i, v_j]$ from v_i to v_j with capacity 1 (or 0 if it is a loop) will be added to the edge-set for N.

(b) A tuple of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i. The elements of the sets N_i must be elements of V. If $N_i = \{u_1, u_2, \ldots, u_r\}$, the edges $[v_i, u_1], \ldots, [v_i, u_r]$ will be added to N, all with capacity 1 (or 0 if they are loops).

(c) A tuple of the form $\langle [v_i, v_j], c \rangle$ where v_i, v_j are vertices in V and c the non-negative capacity of the directed edge $[v_i, v_j]$ added to N.

(d) A sequence $[N_1, N_2, \ldots, N_n]$ of n sets, where N_i will be interpreted as a set of out-neighbours for the vertex v_i. All the edges $[v_i, u_i]$, $u_i \in N_i$, are added to N with capacity 1 (or 0 if they are loops).

In addition to these four basic ways of specifying the *edges* list, the items in *edges* may also be:

(e) An edge e of a graph (or di/multi/multidigraph) or network of order n. If e is an edge of a network H, then it will be added to N with the capacity it has in H. If e is not a network edge, then it will be added to N with capacity 1, or 0 if it is a loop.

(f) An edge-set E of a graph (or di/multi/multidigraph) or network of order n. Every edge e in E will be added to N according to the rule set out for a single edge.

(g) A graph (or di/multi/multidigraph) or network H of order n. Every edge e in H's edge-set is added to N according to the rule set out for a single edge.

(h) A $n \times n$ (0, 1)-matrix A. The matrix A will be interpreted as the adjacency matrix for a digraph H on n vertices and the edges of H will be included among the edges of N with capacity 1 (0 if loops).

(i) A set of

(i) Pairs of the form $[v_i, v_j]$ of vertices in V.

(ii) Tuples of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i.

(iii) A tuple of the form $\langle [v_i, v_j], c \rangle$ where v_i, v_j are vertices in V and c a non-negative capacity.

(iv) Edges of a graph (or di/multi/multidigraph) or network of order n.

(v) Graphs (or di/multi/multidigraphs) or networks of order n.

(j) A sequence of

(i) Tuples of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i.

(ii) A tuple of the form $\langle [v_i, v_j], c \rangle$ where v_i, v_j are vertices in V and c a non-negative capacity.

Example: Network from a Digraph (ex-4c4672)

We construct a network from a digraph, and observe that the edges that are not loops have a capacity of 1:

```
> SetSeed(1, 0);
> n := 5;
> d := 0.2;
> D := RandomDigraph(n, d : SparseRep := true);
> N := Network< n | D >;
> D;
Digraph
```

(continues on next page)

(continued from previous page)

```
Vertex  Neighbours

1       2 1 ;
2       3 2 ;
3       ;
4       5 ;
5       2 ;

> N;
Network
Vertex  Neighbours

1       1 [ 0 ] 2 [ 1 ] ;
2       2 [ 0 ] 3 [ 1 ] ;
3       ;
4       5 [ 1 ] ;
5       2 [ 1 ] ;
```

164.2.1 Magma Output: Printing of a Network

Magma displays a network N in the form of a list of vertices, each accompanied by a list of its outgoing capacitated edges (each followed by the capacity of the edge in brackets). Thus, in the previous example *Network from a Digraph*, it can be verified that all edges have capacity 1 (since the network was constructed from a digraph) except those edges that are loops.

If the network has multiple edges from u to v, then *each* edge from u to v, or rather its end-point v, is printed followed by the capacity of that edge. Also, the end-points in the adjacency list are not ordered and appear in the order in which they were created. The next example illustrates these two points.

Example: Network from Capacity Tuples `(ex-5be897)`

We construct a network from a set of tuples $< [vertex, vertex], capacity >$ and we exhibit a multiple edge.

```
> n := 5;
> C := 5;
> M := 3;
> T := [];
> for i in [1..12] do
>     u := Random(1, n);
>     v := Random(1, n);
>     m := Random(1, M);
>     for j in [1..m] do
>        c := Random(0, C);
>        if u eq v then
>            Append(~T, < [u, u], 0 >);
>        else
>            Append(~T, < [u, v], c >);
>        end if;
>     end for;
> end for;
> T;
[
  <[ 5, 4 ], 1>, <[ 5, 4 ], 3>, <[ 5, 4 ], 2>, <[ 5, 4 ], 1>,
  <[ 5, 4 ], 5>, <[ 1, 3 ], 2>, <[ 1, 3 ], 2>, <[ 5, 5 ], 0>,
  <[ 5, 5 ], 0>, <[ 2, 1 ], 2>, <[ 4, 2 ], 2>, <[ 4, 2 ], 5>,
  <[ 4, 2 ], 1>, <[ 4, 1 ], 3>, <[ 4, 1 ], 4>, <[ 4, 1 ], 3>,
```

(continues on next page)

(continued from previous page)

```
  <[ 2, 3 ], 1>, <[ 2, 3 ], 3>, <[ 4, 3 ], 5>, <[ 4, 3 ], 3>,
  <[ 4, 3 ], 4>, <[ 2, 2 ], 0>, <[ 2, 2 ], 0>, <[ 5, 4 ], 0>,
  <[ 4, 4 ], 0>
]
> N := Network< n | T >;
> N;
Network
Vertex  Neighbours

1       3 [ 2 ] 3 [ 2 ] ;
2       2 [ 0 ] 2 [ 0 ] 3 [ 3 ] 3 [ 1 ] 1 [ 2 ] ;
3       ;
4       4 [ 0 ] 3 [ 4 ] 3 [ 3 ] 3 [ 5 ] 1 [ 3 ] 1 [ 4 ] 1 [ 3 ] 2
        [ 1 ] 2 [ 5 ] 2 [ 2 ] ;
5       4 [ 0 ] 5 [ 0 ] 5 [ 0 ] 4 [ 5 ] 4 [ 1 ] 4 [ 2 ] 4 [ 3 ] 4
        [ 1 ] ;

> Edges(N);
{@ < [1, 3], 6 >, < [1, 3], 7 >, < [2, 1], 10 >, < [2, 2], 22 >,
< [2, 2], 23 >, < [2, 3], 17 >, < [2, 3], 18 >, < [4, 1], 14 >,
< [4, 1], 15 >, < [4, 1], 16 >, < [4, 2], 11 >, < [4, 2], 12 >,
< [4, 2], 13 >, < [4, 3], 19 >, < [4, 3], 20 >, < [4, 3], 21 >,
< [4, 4], 25 >, < [5, 4], 1 >, < [5, 4], 2 >, < [5, 4], 3 >,
< [5, 4], 4 >, < [5, 4], 5 >, < [5, 4], 24 >, < [5, 5], 8 >,
< [5, 5], 9 > @}
```

164.3 Standard Construction for Networks

164.3.1 Subgraphs

The construction of a sub-network is very similar to the construction of a sub-multidigraph (see Subsection *Subgraphs*). Additional flexibility is available for setting edge capacities in the subgraph.

There are two constraints when building a subgraph (see the introduction to Section *Elementary Invariants and Predicates for Multigraphs*). Let N be a network, and H a subgraph of N. Then, given any vertices u, v of H, the edge multiplicity from u to v is no greater in H than it is in N, and the total capacity from u to v in H is no greater than the total capacity from u to v in N. Failure to satisfy these constraints will result in a run-time error when constructing a sub-network.

Assume that we intend to add an edge from u to v in H. Assume also that the total capacity from u to v in N is C_N, that the total capacity from u to v in H is C_H before adding the edge from u to v in H, and that the total capacity from u to v in H is C'_H after adding the edge from u to v in H. In order to satisfy the "capacity constraint" it is enough that $C'_H \leq C_N$, i.e. that $C_H + c \leq C_N$ where c is the capacity of the edge one wants to add in H.

There are two methods for adding an edge from u to v. Firstly, adding the edge $[u, v]$ to H without specifying its capacity in H assumes that the edge $[u, v]$ will be added with capacity C_N. This implies that C_H is zero, since we require that $C_H + C_G \leq C_N$. Secondly, adding the edge $[u, v]$ to H when specifying its capacity as c assumes that the edge $[u, v]$ will be added with capacity c such that $C_H + c \leq C_N$.

Note that the support set, vertex labels and edge labels and weights, if applicable, are transferred from the network to the sub-network.

sub< N | list >: GrphNet, List -> GrphNet, GrphVertSet, GrphEdgeSet

Construct the network H as a subgraph (sub-network) of N. The function returns three values: The network H, the vertex-set V of H; and the edge-set E of H. If N has a support set and/or if N has vertex/edge labels, and/or edge weights then *all* these attributes are transferred to the

subgraph H. Edge capacities are also transferred to H unless they are specifically set as explained below.

The elements of V and of E are specified by the list *list* whose items can be objects of the following types:

(a) A vertex of N. The resulting subgraph will be the subgraph induced on the subset of `VertexSet(N)` defined by the vertices in *list*.

(b) An edge of N. The resulting subgraph will be the subgraph with vertex-set `VertexSet(N)` whose edge-set consists of the edges in *list* subject to the multiplicity and capacity constraints being satisfied.

(c) A pair of $[v_i, v_j]$ of vertices of N. The resulting subgraph will be the subgraph with vertex-set `VertexSet(N)` whose edge-set consists of the edges $[v_i, v_j]$ in *list* whose capacity is assumed to be the total capacity from v_i to v_j in N. The multiplicity and capacity constraints must be satisfied.

(d) A tuple of the form $< [v_i, v_j], c >$ where v_i, v_j are vertices of N and c the non-negative capacity of the edge $[v_i, v_j]$ to be added to H. The resulting subgraph will be the subgraph with vertex-set `VertexSet(N)` whose edge-set consists of the edges as they are given in *list*, subject to the multiplicity and capacity constraints being satisfied.

(e) A set of

(i) Vertices of N.

(ii) Edges of N.

(iii) Pairs of vertices of N.

(iv) Tuples of the form $< [v_i, v_j], c >$ where v_i, v_j are vertices of N and c the non-negative capacity of the edge $[v_i, v_j]$ to be added to H.

Example: Subgraphs of a Network `(ex-48c2fa)`

We start by constructing a network with some multiple edges.

```
> N := Network< 4 |
> < [1, 2], 2 >, < [1, 2], 3 >, < [1, 4], 5 >,
> < [2, 3], 1 >, < [2, 3], 3 >, < [3, 4], 1 >, < [3, 4], 6 > >;
> N;
Network
Vertex  Neighbours

1       4 [ 5 ] 2 [ 3 ] 2 [ 2 ] ;
2       3 [ 3 ] 3 [ 1 ] ;
3       4 [ 6 ] 4 [ 1 ] ;
4       ;

> V := VertexSet(N);
> E := EdgeSet(N);
```

We construct a subgraph H of N induced by some of N's vertices and we obtain the mapping from the vertices of N to the vertices of H and vice-versa.

```
> H := sub< N | V!1, V!3, V!4 >;
> assert IsSubgraph(N, H);
> H;
Network
Vertex  Neighbours
```

(continues on next page)

(continued from previous page)

```
1       3 [ 5 ] ;
2       3 [ 1 ] 3 [ 6 ] ;
3       ;

> V!VertexSet(H)!1, VertexSet(H)!V!1;
1 1
> V!VertexSet(H)!2, VertexSet(H)!V!3;
3 2
> V!VertexSet(H)!3, VertexSet(H)!V!4;
4 3
```

The next statements illustrate the "capacity constraint": That is, given any pair $[u, v]$ of vertices of H, H a subgraph of N, the total capacity from u to v in H can not be greater than the total capacity from u to v in N. The subgraph constructor will fail whenever this rule cannot be satisfied. We give a few examples below.

```
> Edges(N);
{@ < [1, 2], 1 >, < [1, 2], 2 >, < [1, 4], 3 >, < [2, 3], 4 >, < [2, 3], 5 >,
< [3, 4], 6 >, < [3, 4], 7 > @}
> E.1, E.2;
< [1, 2], 1 > < [1, 2], 2 >
> Capacity(E.1);
2
> Capacity(E.2);
3
> Capacity(V!1, V!2);
5
>
> H := sub< N | E.1, E.1 >;
> H;
Network
Vertex  Neighbours

1       2 [ 2 ] 2 [ 2 ] ;
2       ;
3       ;
4       ;
```

Adding twice the edge $E.1$ to H is a valid operation since the resulting total capacity from 1 to 2 in H is 4 while it is 5 in N.

```
> H := sub< N | E.2, E.2 >;

>> H := sub< N | E.2, E.2 >;
             ^
Runtime error in sub< ... >: RHS argument 2 - Edge multiplicity and capacity
not compatible with subgraph constructor
>
```

Adding twice the edge $E.2$ (which has capacity 3) to H would have resulted in the total capacity from 1 to 2 in H to be 6, while it is 5 in N.

```
> H := sub< N | E!< [1, 2], 1 >, E!< [1, 2], 1 > >;
> H;
Network
Vertex  Neighbours
```

(continues on next page)

(continued from previous page)

```
1       2 [ 2 ] 2 [ 2 ] ;
2       ;
3       ;
4       ;
```

This succeeded since the total capacity from 1 to 2 in H is now 4.

```
> H := sub< N | E!< [1, 2], 2 >, E!< [1, 2], 2 > >;

>> H := sub< N | E!< [1, 2], 2 >, E!< [1, 2], 2 > >;
             ^
Runtime error in sub< ... >: RHS argument 2 - Edge multiplicity and capacity
not compatible with subgraph constructor
>
```

Again, this operation failed as it would have resulted in the total capacity from 1 to 2 in H to be 6.

```
> H := sub< N | < [ V!1, V!2 ], 2 >, < [ V!1, V!2 ], 2 > >;
> H;
Network
Vertex  Neighbours

1       2 [ 2 ] 2 [ 2 ] ;
2       ;
3       ;
4       ;
```

This operation is valid since the total capacity from 1 to 2 in H is now 4.

```
> H := sub< N | < [ V!1, V!2 ], 2 >, < [ V!1, V!2 ], 4 > >;

>> H := sub< N | < [ V!1, V!2 ], 2 >, < [ V!1, V!2 ], 4 > >;
             ^
Runtime error in sub< ... >: RHS argument 2 - Tuple must be <[vertex,
vertex], capacity> with total edge multiplicity and capacity compatible with
subgraph constructor
>
```

This operation cannot succeed as the total capacity from 1 to 2 in H would be 6.

```
> H := sub< N | [ V!1, V!2 ] >;
> H;
Network
Vertex  Neighbours

1       2 [ 5 ] ;
2       ;
3       ;
4       ;
```

Adding the edge $[1, 2]$ without specifying its capacity implies that an edge from 1 to 2 is added with capacity 5 to H, which is the total capacity from 1 to 2 in N.

```
> H := sub< N | [ V!1, V!2 ], [ V!1, V!2 ] >;

>> H := sub< N | [ V!1, V!2 ], [ V!1, V!2 ] >;
```

(continues on next page)

(continued from previous page)

```
           ^
Runtime error in sub< ... >: RHS argument 2 - Sequence must be
[vertex, vertex] with vertices of the LHS and must be unique
>
```

Adding twice the edge $[1, 2]$ without specifying its capacity would have resulted in the total capacity from 1 to 2 in H to be 10. This operation cannot succeed. Finally, let us illustrate the edge multiplicity constraint.

```
> H := sub< N | E.4, E.4, E.4 >;

>> H := sub< N | E.4, E.4, E.4 >;
           ^
Runtime error in sub< ... >: RHS argument 3 - Edge multiplicity and capacity
not compatible with subgraph constructor
>
```

Although the above statement satisfies the capacity constraint

```
> Capacity(E.4);
1
> Capacity(V!InitialVertex(E.4), V!TerminalVertex(E.4));
4
```

it cannot succeed since the edge multiplicity constraint is violated.

```
> EdgeMultiplicity(V!InitialVertex(E.4), V!TerminalVertex(E.4));
2
```

164.3.2 Incremental Construction: Adding Edges

Almost all the functions to add or remove either vertices or edges that are available for multidigraphs also apply to networks; they are not listed here, see Section *Incremental Construction of Multigraphs* for details.

The only exception is the function `AddEdge(G, u, v, l)` which differs for multidigraphs and networks. It is replaced by the functions `AddEdge(G, u, v, c)` and `AddEdge(G, u, v, c, l)` which are specialised functions for adding capacitated edges to networks. There are a few more such specialised functions which are listed below, they all concern adding edges to a network.

Note that whenever an edge is added to a network using the general multidigraph functions, which do not allow specifying an edge capacity, the edge to be added is *always taken to have capacity 1 (or 0 if a loop)*.

`N + < [ u, v ], c >: GrphNet, < [ GrphVert, GrphVert ], RngIntElt > -> GrphNet, GrphEdge`

Given two vertices u and v of a network N, and c a non-negative integer, adds an edge from u to v with capacity c. The support and edge capacities are retained. This function returns two values: The modified network, and the edge newly created (added). This feature is especially useful when adding parallel edges.

`N + { < [ u, v ], c > }: GrphNet, { < [ GrphVert, GrphVert ], RngIntElt > } -> GrphNet`

`N + [ < [ u, v ], c > ]: GrphNet, [ < [ GrphVert, GrphVert ], RngIntElt > ] -> GrphNet`

Given tuples of the form $< [u, v], c >$, u and v vertices of the network N and c a non-negative integer, adds the edges from u to v with capacity c. Tuples can be contained in a set or a sequence; the latter is useful when dealing with duplicates. The support and edge capacities are retained.

`N +:= < [ u, v ], c >: GrphNet, < [ GrphVert, GrphVert ], RngIntElt >`

`N +:= { < [ u, v ], c > }: GrphNet, { < [ GrphVert, GrphVert ], RngIntElt > }`

`N +:= [ < [ u, v ], c > ]: GrphNet, [ < [ GrphVert, GrphVert ], RngIntElt > ]`

The procedural versions of the previous three functions. Tuples can be contained in a set or a sequence; the latter is useful when dealing with duplicates.

`AddEdge(N, u, v, c): GrphNet, GrphVert, GrphVert, RngIntElt -> GrphNet, GrphEdge`

Given two vertices u and v of the network N, and c a non-negative integer, adds an edge from u to v with capacity c. The support and edge capacities are retained. This function returns the modified network and the newly created edge. This feature is especially useful when adding parallel edges.

`AddEdge(N, u, v, c, l): GrphNet, GrphVert, GrphVert, RngIntElt, . -> GrphNet, GrphEdge`

Given two vertices u and v of the network N, c a non-negative integer, and a label l, adds an edge from u to v with capacity c and label l. The support and edge capacities are retained. This function returns the modified network and the newly created edge. This feature is especially useful when adding parallel edges.

`AddEdges(N, S): GrphNet, { < [ GrphVert, GrphVert ], RngIntElt > } -> GrphNet`

`AddEdges(N, S): GrphNet, [ < [ GrphVert, GrphVert ], RngIntElt > ] -> GrphNet`

Given a network N and a set or a sequence S of tuples, this function includes the edges specified in S. The tuples must be of the form $< [u, v], c >$, where u and v vertices of N and c a non-negative integer. The support and existing vertex and edge decorations are retained.

`AddEdge(~N, u, v, c): GrphNet, GrphVert, GrphVert, RngIntElt`

`AddEdge(~N, u, v, c, l): GrphNet, GrphVert, GrphVert, RngIntElt, .`

`AddEdges(~N, S): GrphNet, { < [ GrphVert, GrphVert ], RngIntElt > }`

`AddEdges(~N, S): GrphNet, [ < [ GrphVert, GrphVert ], RngIntElt > ]`

Procedural versions of previous functions adding edges to a network. Tuples can be contained in a set or a sequence; the latter is useful when dealing with duplicates.

164.3.3 Union of Networks

It is possible to construct a new network from the union of two networks. For more details, we refer the reader to Subsection *Unions of Multigraphs*.

164.4 Maximum Flow and Minimum Cut

All the functions described in this section apply to general graphs whose edges are given a capacity, that is, networks. If the graph under consideration is not capacitated, then all its edges are assumed to have capacity one. To assign capacities to the edges of a graph, see Subsection *Edge Decorations*. To create a Magma network object (with type `GrphNet`) see Section *Construction of Networks*.

The fundamental network flow problem is the minimum cost flow problem, that is, determining a maximum flow at minimum cost from a specified source to a specified sink. Specializations of this problem are the shortest path problem (where there is no capacity constraint) which is covered in Section *Distances, Shortest Paths, and Minimum Weight Trees*, and the maximum flow problem (where there is no cost constraint). Some of the related problems are the minimum spanning tree problem, the matching problem (a pairing of the edges of the graph according to some criteria), and the multicommodity flow problem (where arcs may carry several flows of different nature). For a comprehensive monograph on network problems, their implementation, and applications, see [AMO93].

Let G be a general graph or multigraph whose edges are capacitated. If G is undirected then consider any edge $\{u, v\}$ with capacity c as being equivalent to two directed edges $[u, v]$ and $[v, u]$, both with capacity c. We may thus assume without loss of generality that G is directed, and from now on G is called a network. Let V and E be G's vertex-set and edge-set respectively, and for every edge $[u, v]$ in G, denote its capacity by $c(u, v)$. If there is no edge $[u, v]$ in E we assume that $c(u, v) = 0$. By convention the capacity of an edge is a non-negative integer.

Distinguish two vertices of G, the *source* s and the *sink* t. A *flow* in G is an integer-valued function $f : V \times V \to \mathbb{Z}$ that satisfies the following properties:

(i) Capacity constraint: $\forall u, v \in V$, $f(u, v) \leq c(u, v)$.

(ii) Skew symmetry: $\forall u, v \in V,\ f(u,v) = -f(v,u)$.

(iii) Flow conservation: $\forall u \in V \setminus \{s,t\},\ \sum_{v \in V} f(u,v) = 0$.

Note that the flow could have been defined as a real-valued function if the edge capacity had been defined as real-valued.

The quantity $f(u,v)$, which can be positive or negative, is called the *net flow* from vertex u to vertex v. The *value* of a flow f is defined as $F = \sum_{v \in V} f(s,v)$, that is, the total net flow out of the source.

In the *maximum flow problem* we wish to find a flow of maximum value from s to t. A *cut* of the network G with source s and sink t is a partition of V into S and T such that $s \in S$ and $t \in T$. The *capacity* of the cut $c(S)$ determined by S is the sum of the capacity $c(u,v)$ of the edges $[u,v]$ such that $u \in S$ and $v \in T$. It is easy to see that $F \leq c(S)$, that is, the value of the flow from s into t cannot be larger than the capacity of any cut defined by s and t. In fact, $F = c(S)$ if and only if F is a maximum flow and S is a cut of minimum capacity.

We have implemented two algorithms for finding the maximum flow in a network G from a source s to a sink t. One is the Dinic algorithm, the other is an instance of the generic push-relabel algorithm. We have already encountered both algorithms in Subsections *Maximum Matching in Bipartite Graphs* and *General Vertex and Edge Connectivity in Graphs and Digraphs*.

The Dinic algorithm

The Dinic algorithm consists of two phases. In the first phase, one constructs a layered network which consists of all the "useful" edges of G: A useful edge $[u,v]$ has the property that $f(u,v) < c(u,v)$. The first and the last layer of this layered network always contain s and t respectively. If such a layered network cannot be constructed then the flow in G is maximum and the algorithm ends.

In the second phase of the Dinic algorithm, one finds a maximal flow by constructing paths from s to t (using a depth-first search) in the layered network. Note that a maximal flow may not be maximum: A maximal flow satisfies the condition that for every path P from s to t in the layered network, there is at least a *saturated* edge in P. A saturated edge is one for which its flow equals its capacity. For more details on the Dinic algorithm, see [Eve79].

The Dinic algorithm has a theoretical complexity of $\mathrm{O}(|V|^2|E|)$, which can be improved to $\mathrm{O}(|E|^{3/2})$ if G is a zero-one network (the capacities of the edges are either 0 or 1), or $\mathrm{O}(|V|^{2/3}|E|)$ if G is a zero-one network with no parallel edges. Prior to the advent of the push-relabel method (which we describe below), the Dinic algorithm was shown to be superior in practice to other methods. The Magma implementation of the Dinic algorithm only outperforms the push-relabel method for very sparse networks (a small number of edges relative to the number of vertices) whose edges have a small capacity and where the maximum flow is small. In other words, the Dinic method performs best on zero-one and very sparse networks.

The push-relabel method

The generic push-relabel algorithm does not construct a flow by constructing paths from s to v. Rather, it starts by pushing the maximum possible flow out from the source s into the neighbours of s, then pushing the excess flow at those vertices into their own neighbours. This is repeated until all vertices of G except s and t have an excess flow of zero (that is, the flow conservation property is satisfied). Of course this might mean that some flow is pushed back into s.

At initialisation all vertices are given a height of 0 except s which will have height $|V|$, and flow is pushed out of s. The maximum flow that can be pushed out of s is $\sum_{v \in V} c(s,v)$. The idea is to push flow into vertices only in a "downward" manner, that is, from a vertex with height $h+1$ into a vertex with height h. If a vertex in $V \setminus \{s,t\}$ has height h, it will be relabelled if it has an excess flow and none of its neighbours to which some flow could be pushed has height $h-1$. Its new label (height) will be $l+1$ where l is the height of the lowest labelled neighbour towards which flow can be pushed. Flow may only be pushed along an edge $[u,v]$ while the capacity constraint $f(u,v) \leq c(u,v)$ is satisfied.

So the general notion is of pushing flow "downward" and "discharging" the vertices in $V \setminus \{s,t\}$ from all non-zero excess flow they may have accumulated, a process which may require "lifting" some vertices so that discharging can proceed. It is easy to show that the height of any vertex in $V \setminus \{s,t\}$ is bounded by $2|V|-1$, so the algorithm must terminate. When this happens, the flow is at a maximum.

The theoretical complexity of the generic push-relabel algorithm is $O(|V|^2|E|)$ which can be improved to at least $O(|V||E|\log(|V|^2/|E|)$ thanks to some heuristics.

One commonly used heuristic is the *global relabelling* procedure whereby the vertices in $V \setminus \{s,t\}$ are pre-labelled according to their distance from the sink. Global relabelling may occur at specified intervals during execution. Another very useful heuristic is *gap relabelling* which involves recognising those vertices that have become unreachable from the sink and labelling them accordingly so that they won't be chosen during later execution. Finally, another heuristic involves the order in which the vertices to be discharged are processed.

When G is a one-zero network, choosing the vertex with smallest height works best, while choosing the vertex with largest height is recommended for general networks. These heuristics are fully described and analysed by B.V. Cherkassky and A.V. Goldberg *et al.* in [CGM+98] and [CG97]. The Magma implementation incorporates all the above heuristics together with some new ones.

As a rule, whenever the push-relabel algorithm is applied to a zero-one network (such as occurs when computing the connectivity of a graph in *General Vertex and Edge Connectivity in Graphs and Digraphs* or finding a maximum matching in a bipartite network in *Maximum Matching in Bipartite Graphs*) the vertex with smallest height is chosen as the next vertex to be processed. When the algorithm is used to compute a maximum flow in a network (as below) the vertex with largest height is chosen.

For the two functions `MinimumCut` and `MaximumFlow` described below, the `PushRelabel` algorithm is used. It is only in the case of a very sparse zero-one network that `Dinic` may outperform `PushRelabel`.

`MinimumCut(s, t : parameters): GrphVert, GrphVert -> SeqEnum, RngIntElt`

```
Al: MonStgElt                    Default: "PushRelabel"
```

Given vertices s and t of a network G, this function returns the subset S defining a minimum cut $\{S,T\}$ of V, where $s \in S$, $t \in T$, corresponding to the maximum flow F from s to t. The subset S is returned as a sequence of vertices of G. The maximum flow F is returned as a second value. The parameter `Al` enables the user to select the algorithm to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`MinimumCut(Ss, Ts : parameters): [ GrphVert ], [ GrphVert ] -> SeqEnum, RngIntElt`

```
Al: MonStgElt                    Default: "PushRelabel"
```

Given two sequences S_s and T_s of vertices of a network G, this function returns the subset S defining a minimum cut $\{S,T\}$ of V, such that $S_s \subseteq S$ and $T_s \subseteq T$, and which corresponds to the maximum flow F from the vertices in S_s to the vertices in T_s. The subset S is returned as a sequence of vertices of G. The maximum flow F is returned as a second value. The parameter `Al` enables the user to select the algorithm to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`MaximumFlow(s, t : parameters): GrphVert, GrphVert -> RngIntElt, SeqEnum`

```
Al: MonStgElt                    Default: "PushRelabel"
```

Given vertices s and t of a network G, this function returns the maximum flow F from s to t. The subset S defining a minimum cut $\{S,T\}$ of `VertexSet(N)`, $s \in S$, $t \in T$, corresponding to F is returned as the second value. The subset S is returned as a sequence of vertices of G. The parameter `Al` enables the user to select the algorithm to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`MaximumFlow(Ss, Ts : parameters): [ GrphVert ], [ GrphVert ] -> RngIntElt, SeqEnum`

```
Al: MonStgElt                    Default: "PushRelabel"
```

Given two sequences S_s and T_s of vertices of a network G, this function returns the maximum flow F from the vertices in S_s to the vertices in T_s. The subset S defining a minimum cut $\{S,T\}$ of `VertexSet(N)`, $S_s \subseteq S$ and $T_s \subseteq T$, corresponding to F is returned as the second value. The

subset S is returned as a sequence of vertices of G. The parameter `Al` enables the user to select the algorithm to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`Flow(e): GrphEdge -> RngIntElt`

Given an edge e in a network G, this function returns the flow of the edge e as an integer. In this instance we require that the edges of G be explicitly assigned capacities (see Subsection *Edge Decorations* or Section *Construction of Networks*). Edges of G will carry a flow only if a flow has been constructed from a source to a sink in G. If no such flow has yet been constructed, all edges will have zero flow.

`Flow(u, v): GrphVert, GrphVert -> RngIntElt`

Let u and v be adjacent vertices of a network G whose edges have been explicitly assigned capacities (see Subsection *Edge Decorations* or Section *Construction of Networks*). `Flow(u, v)` returns the total net flow from u to v as an integer. The total net flow from u to v is defined as the total outgoing flow from u into v minus the total ingoing flow into u from v. Thus the flow satisfies the skew symmetry `Flow(u, v)` = - `Flow(v, u)`.

Edges of G will carry a flow only if a flow has been constructed from a source to a sink in G. If no such flow has yet been constructed, all edges will have zero flow.

Example: Bipartite Matching via Maximum Flow (ex-00952b)

We illustrate the maximum flow algorithm by applying it to find a maximum matching in a bipartite graph. This example exactly replicates the implementation of the maximum matching algorithm (Section *Maximum Matching in Bipartite Graphs*).

We construct a bipartite graph.

```
> G := Graph< 7 | [ {5, 7}, {6, 7}, {4, 5, 6},
> {3}, {1, 3}, {2, 3}, {1, 2} ] : SparseRep := true >;
> assert IsBipartite(G);
> P := Bipartition(G);
> P;
[
  { 1, 2, 3 },
  { 4, 5, 6, 7 }
]
```

We add two extra vertices to G, the source s and the sink t and we construct three sets of pairs of vertices. The first contains all the pairs $[s, u]$ where $u \in P[1]$, the second contains all the pairs $[u, t]$ where $u \in P[2]$, and the third contains all the pairs $[u, v]$ where $u \in P[1]$ and $v \in P[2]$. From these sets we construct a multidigraph with capacitated edges.

```
> G +:= 2;
> G;
Graph
Vertex  Neighbours

1       7 5 ;
2       7 6 ;
3       6 5 4 ;
4       3 ;
5       3 1 ;
6       3 2 ;
7       2 1 ;
8       ;
9       ;

> s := Order(G)-1;
> t := Order(G);
```

(continues on next page)

(continued from previous page)

```
>
> E1 := { [s, Index(u)] : u in P[1] };
> E2 := { [Index(u), t] : u in P[2] };
> E3 := { [Index(u), Index(v)] : u in P[1], v in P[2] };
>
> N := MultiDigraph< Order(G) | E1, E2, E3 >;
> N;
Multidigraph
Vertex  Neighbours

1       7 6 5 4 ;
2       5 4 7 6 ;
3       7 6 5 4 ;
4       9 ;
5       9 ;
6       9 ;
7       9 ;
8       1 3 2 ;
9       ;

> E := EdgeSet(N);
> for e in E do
>     AssignCapacity(~N, e, 1);
> end for;
```

First note that all the edges of N have been assigned capacity 1. Also, the edges of N are those connecting s to all the vertices in $P[1]$, those connecting all the vertices of $P[2]$ to t, and those connecting all the vertices of $P[1]$ to all the vertices of $P[2]$.

We construct a maximum flow F from s to t and we check that the capacity of the cut is indeed F.

```
> V := VertexSet(N);
> F := MaximumFlow(V!s, V!t);
> S := MinimumCut(V!s, V!t);
> F;
3
> S;
[ 8 ]
>
> c := 0;
> for u in S do
>     for v in OutNeighbours(u) do
>        if not v in S then
>           c +:= Capacity(u, v);
>           assert Capacity(u, v) eq Flow(u, v);
>        end if;
>     end for;
> end for;
> assert c eq F;
```

We now exhibit a matching in G. It will be given by the edges in E_3 which are saturated, that is, whose flow is 1.

```
> M := [];
> for e in E3 do
>     u := V!e[1];
```

(continues on next page)

(continued from previous page)

```
>     v := V!e[2];

>     if Flow(u, v) eq 1 then
>         E := Edges(u, v);
>         assert #E eq 1;
>         Append(~M, EndVertices(E[1]));
>     end if;
> end for;
> assert #M eq F;
> M;
[
  [ 1, 7 ],
  [ 3, 5 ],
  [ 2, 4 ]
]
```

Note that the statement

```
>        assert #E eq 1;
```

makes sense as the network N has no parallel edges.

We end the example by showing that the capacity constraint and the skew symmetry is satisfied by all edges in the network.

```
> for e in EdgeSet(N) do
>     u := EndVertices(e)[1];
>     v := EndVertices(e)[2];
>
>     assert Flow(u, v) le Capacity(u, v);
>     assert Flow(u, v) eq  -Flow(v, u);
> end for;
```

Also, flow conservation holds for all vertices in $V \setminus \{s, t\}$.

```
> s := V!s;
> t := V!t;
> for u in V do
>     if not u eq s and not u eq t then
>         f := 0;
>         for v in OutNeighbours(u) do
>             E := Edges(u, v);
>             f +:= Flow(E[1]);
>         end for;
>         for v in InNeighbours(u) do
>             E := Edges(v, u);
>             f -:= Flow(E[1]);
>         end for;
>
>         assert f eq 0;
>     end if;
> end for;
```

164.5 Bibliography

- [AMO93] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows, Theory, Algorithms, and Applications*. Prentice Hall, 1993.
- [CG97] B.V. Cherkassky and A.V. Golberg. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19:390–410, 1997.
- [CGM+98] B.V. Cherkassky, A.V. Golberg, Paul Martin, J.C. Setubal, and J. Stolfi. Augment or Push? A Computational Study of Bipartite Matching and Unit Capacity Flow Algorithms. Technical Report, NEC Research Institute, 1998.
- [Eve79] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

References

[AMO93] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows, Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[BM01] J. Boyer and W. Myrvold. Simplified O(n) Planarity Algorithms. submitted, 2001.

[Bre79] D. Brelaz. New Methods to Color the Vertices of a Graph. *Communications of the ACM*, 22(9):251–256, 1979.

[BK73] C. Bron and J. Kerbosch. Finding All Cliques of an Undirected Graph. *Communications of the ACM 9*, 16(9):575–577, 1973.

[CG97] B.V. Cherkassky and A.V. Golberg. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19:390–410, 1997.

[CGM+98] B.V. Cherkassky, A.V. Golberg, Paul Martin, J.C. Setubal, and J. Stolfi. Augment or Push? A Computational Study of Bipartite Matching and Unit Capacity Flow Algorithms. Technical Report, NEC Research Institute, 1998.

[Chr75] N. Christofides. *Graph Theory, An Algorithm Approach*. Academic Press, 1975.

[Die00] Reinhard Diestel. *Graph Theory, Second Edition*. Springer, 2000.

[Eve79] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

[Ful97b] William Fulton. *Young Tableaux*. Cambridge University Press, Cambridge, 1997. ISBN 0-521-56144-2; 0-521-56724-6.

[GNW84] Curtis Greene, Albert Nijenhuis, and Herbert S. Wilf. Another probabilistic method in the theory of Young tableaux. *J. Combin. Theory Ser. A*, 37(2):127–135, 1984.

[GM01] C. Gutwenger and P. Mutzel. A Linear Time Implementation of SPQR-Trees. In J. Marks, editor, *Graph Drawing 2000*, volume 1984 of LNCS, pages 70–90. Springer-Verlag, 2001.

[Hal86] Marshall Hall. *Combinatorial Theory*. New York: Wiley, 2nd edition, 1986.

[HT73] J.E. Hopcroft and R.E. Tarjan. Dividing a Graph into Triconnected Components. *SIAM J. Comput.*, 2(3):135–158, 1973.

[KKL92] Adalbert Kerber, Axel Kohnert, and Alain Lascoux. SYMMETRICA, an object oriented computer-algebra system for the symmetric group. *J. Symbolic Comp.*, 14(2-3):195–203, 1992. http://www.mathe2.uni-bayreuth.de/axel/symneu_engl.html.

[LS85] Alain Lascoux and Marcel-Paul Schützenberger. Schubert polynomials and the Littlewood-Richardson rule. *Lett. Math. Phys.*, 10(2-3):111–124, 1985.

[Lor89] P. Lorimer. The construction of Tutte's 8-cage and the Conder graph. *J. of Graph Theory*, 13(5):553–557, 1989.

[Mac95] I. G. Macdonald. *Symmetric functions and Hall polynomials*. The Clarendon Press Oxford University Press, New York, second edition, 1995. ISBN 0-19-853489-2. With contributions by A. Zelevinsky, Oxford Science Publications.

[McK81] B. D. McKay. Practical Graph Isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[McK98] B. D. McKay. Isomorph-free exhaustive generation. *J. Algorithms*, 26:306–324, 1998.

[MP] B. D. McKay and A. Piperno. nauty and Traces User's Guide (Version 2.6). http://pallini.di.uniroma1.it/nug26.pdf.

[Mui60] Thomas Muir. *A treatise on the theory of determinants*. Dover Publications Inc., New York, 1960.

[MPW] W. Myrvold, T. Prsa, and N. Walker. A Dynamic Programming Approach for Timing and Designing Clique Algorithms. available at http://www.csr.uvic.ca/ wendym/.

[TCR90] C.E. Leiserson T.H. Cormen and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

Index

Symbols

+ a (*operation*), 36
# B (*operation*), 57, 60
# D (*operation*), 135
# P (*operation*), 57
# S (*operation*), 92
# u (*operation*), 9
# w (*operation*), 7
& join S (*operation*), 159
- a (*operation*), 36

A

A ! f (*operation*), 32
A ! m (*operation*), 33
A ! r (*operation*), 33
a * b (*operation*), 36
a *:= b (*operation*), 36
a + b (*operation*), 36
a +:= b (*operation*), 36
A . i (*operation*), 31
A . P (*operation*), 31
a - b (*operation*), 36
a -:= b (*operation*), 36
a ^ k (*operation*), 36
a ~ b (*operation*), 37
A ` PrintStyle (*attribute*), 35
Action(A, Y) (*function*), 129
Action(G, Y) (*function*), 69
ActionImage(A, Y) (*function*), 129
ActionImage(G, Y) (*function*), 69
ActionKernel(A, Y) (*function*), 129
ActionKernel(G, Y) (*function*), 69
AddEdge(~G, u, v) (*procedure*), 97, 156
AddEdge(~G, u, v, c) (*procedure*), 156
AddEdge(~G, u, v, c, l) (*procedure*), 156
AddEdge(~G, u, v, l) (*procedure*), 97, 156
AddEdge(~N, u, v, c) (*procedure*), 183
AddEdge(~N, u, v, c, l) (*procedure*), 183
AddEdge(G, u, v) (*function*), 97, 156
AddEdge(G, u, v, c) (*function*), 156
AddEdge(G, u, v, c, l) (*function*), 156
AddEdge(G, u, v, l) (*function*), 97, 156
AddEdge(N, u, v, c) (*function*), 183
AddEdge(N, u, v, c, l) (*function*), 183
AddEdges(~G, S) (*procedure*), 97, 157
AddEdges(~G, S, L) (*procedure*), 97, 157
AddEdges(~N, S) (*procedure*), 183
AddEdges(G, S) (*function*), 97, 157
AddEdges(G, S, L) (*function*), 97, 157
AddEdges(N, S) (*function*), 183
AddVertex(~G) (*procedure*), 96, 155
AddVertex(~G, l) (*procedure*), 96, 155
AddVertices(~G, n) (*procedure*), 96, 155
AddVertices(~G, n, L) (*procedure*), 96, 155
AdjacencyMatrix(G) (*function*), 113
AllCliques(G : parameters) (*function*), 117
AllCliques(G, k : parameters) (*function*), 117
AllCliques(G, k, m : parameters) (*function*), 117
Alldeg(G, n) (*function*), 105, 162, 164
AllPairsShortestPaths(G : parameters) (*function*), 172
AllParallelClasses(D) (*function*), 63
AllParallelisms(D) (*function*), 63
AllResolutions(D) (*function*), 62
AllResolutions(D, λ) (*function*), 62
AssignCapacities(~G, D) (*procedure*), 147
AssignCapacities(~G, S, D) (*procedure*), 147
AssignCapacity(~G, e, c) (*procedure*), 147
AssignEdgeLabels(~G, D) (*procedure*), 147
AssignLabel(~G, e, l) (*procedure*), 147
AssignLabel(~G, u, l) (*procedure*), 146
AssignLabels(~G, S, D) (*procedure*), 147
AssignLabels(~G, S, L) (*procedure*), 146
AssignVertexLabels(~G, L) (*procedure*), 146
AssignWeight(~G, e, w) (*procedure*), 147
AssignWeights(~G, D) (*procedure*), 147
AssignWeights(~G, S, D) (*procedure*), 147
Aut(D) (*function*), 67
Automorphism Group and Canonical Graph (*example*), 126
Automorphism Group of a Hadamard Design (*example*), 67
Automorphism Group of a Witt Design (*example*), 69
AutomorphismGroup(D) (*function*), 66
AutomorphismGroup(G : parameters) (*function*), 122
AutomorphismGroup(M : parameters) (*function*), 71
AutomorphismGroupStabilizer(D, k) (*function*), 67
Automorphisms of a Fano Polytope Matrix (*example*), 71
Automorphisms of the Clebsch Graph (*example*), 129
AutomorphismSubgroup(D) (*function*), 67

B

B ! S (*operation*), 53
B . i (*operation*), 52
Ball(u, n) (*function*), 112
BaseRing(L) (*function*), 34
Bell(n) (*function*), 2
BernoulliApproximation(n) (*function*), 2
BernoulliNumber(n) (*function*), 2
BernoulliPolynomial(n) (*function*), 2
BFSTree(u) (*function*), 114, 167
Bicomponents(G) (*function*), 107, 164
Binomial(n, r) (*function*), 1
Bipartite Matching via Maximum Flow (*example*), 186
BipartiteGraph(m, n) (*function*), 87
Bipartition(G) (*function*), 105, 162
Block(D, i) (*function*), 52
Block(D, p, q) (*function*), 60
BlockDegree(D) (*function*), 58
BlockDegree(D, B) (*function*), 60
BlockDegrees(D) (*function*), 58
BlockGraph(D) (*function*), 70, 101
BlockGroup(D) (*function*), 67
Blocks(D) (*function*), 57
BlockSet(D) (*function*), 52
BlockSize(D) (*function*), 58
BlockSize(D, B) (*function*), 60
BlockSizes(D) (*function*), 58
BranchVertexPath(u,v) (*function*), 114
BreadthFirstSearchTree(u) (*function*), 114, 167

C

CanonicalGraph(G) (*function*), 126
Capacities(E) (*function*), 148
Capacities(S) (*function*), 148
Capacity(e) (*function*), 148
CartesianProduct(G, H) (*function*), 99
Catalan(n) (*function*), 1
Category(f) (*function*), 35
Category(L) (*function*), 34
Cayley and Schreier Multigraphs of a Group (*example*), 154
Cayley Graph of a Group (*example*), 150
CayleyGraph(A : parameter) (*function*), 100
CayleyMultiGraph(A : parameter) (*function*), 153
Change of Alphabet (*example*), 39
ChangeSupport(~G, S) (*procedure*), 86, 143
ChangeSupport(G, S) (*function*), 85, 143
ChangGraphs() (*function*), 102
Changing Money with Restricted Partitions (*example*), 6
Changing the Basis of a Symmetric Function (*example*), 33
Character Table of the Symmetric Group (*example*), 44
CharacteristicPolynomial(G) (*function*), 103
Characters Induced from Young Subgroups (*example*), 40
Checking Whether a Tableau Is Standard (*example*), 19
Chromatic Number (*example*), 115
ChromaticIndex(G) (*function*), 115
ChromaticNumber(G) (*function*), 114
ChromaticPolynomial(G) (*function*), 115
Classes(D) (*function*), 133
ClebschGraph() (*function*), 102
CliqueNumber(G : parameters) (*function*), 117
Cliques of a Graph (*example*), 118
ClosureGraph(P, G) (*function*), 101
Code of the Witt Design (*example*), 70
Coefficient(s, p) (*function*), 37
CoefficientRing(L) (*function*), 34
Coercing Symmetric Polynomials (*example*), 32
Coercing Vertices and Edges of a Digraph (*example*), 92
Column(t, j) (*function*), 17
ColumnLength(t, j) (*function*), 18
Columns(t) (*function*), 18
ColumnSkewLength(t, j) (*function*), 18
ColumnWord(t) (*function*), 19
Complement(D) (*function*), 54
Complement(G) (*function*), 98
Complete and Random Digraphs (*example*), 88
CompleteDigraph(n) (*function*), 88
CompleteGraph(n) (*function*), 87
CompleteUnion(G, H) (*function*), 99
Component(u) (*function*), 107, 164, 165
Components(G) (*function*), 106, 164
Conjugate(t) (*function*), 20
ConjugatePartition(P) (*function*), 17
ConnectionNumber(D, p, B) (*function*), 60
Connectivity of a Multigraph (*example*), 166
Constructing a Digraph (*example*), 85
Constructing a Multidigraph (*example*), 142
Constructing a Multigraph (*example*), 141
Constructing the Fano Plane (*example*), 50
Constructing the Grotzsch Graph (*example*), 98
Constructing the Petersen Graph (*example*), 82
Content(t) (*function*), 19
Content(u) (*function*), 9
Content(w) (*function*), 7
Contract(e) (*function*), 98, 158
Contract(S) (*function*), 98, 158
Contract(u, v) (*function*), 98, 158
Contraction of a Witt Design (*example*), 56
Contraction(D, b) (*function*), 54
Contraction(D, p) (*function*), 54
Converse(G) (*function*), 102, 160
Converting Incidence Structures (*example*), 65
Counting Single-Column Tableaux (*example*), 27

Counting Standard Tableaux (*example*), 26
Covalence(D, S) (*function*), 58
Covalence(D, s) (*function*), 58
Creating Tableaux over the Integers (*example*), 13
CutVertices(G) (*function*), 107, 164

D

D eq E (*operation*), 66
D ne E (*operation*), 66
Database of Strongly Regular Graphs (*example*), 133
Decomposing and Rebuilding an Element (*example*), 38
Degree(s) (*function*), 38
Degree(u) (*function*), 105, 162, 163
DegreeRange(D) (*function*), 76
Degrees(D) (*function*), 76
DegreeSequence(G) (*function*), 105, 106, 163, 164
DeleteCapacities(~G) (*procedure*), 149
DeleteCapacities(~G, S) (*procedure*), 149
DeleteCapacity(~G, e) (*procedure*), 149
DeleteEdgeLabels(~G) (*procedure*), 149
DeleteLabel(~G, e) (*procedure*), 149
DeleteLabel(~G, u) (*procedure*), 146
DeleteLabels(~G, S) (*procedure*), 146, 149
DeleteVertexLabels(~G) (*procedure*), 146
DeleteWeight(~G, e) (*procedure*), 149
DeleteWeights(~G) (*procedure*), 149
DeleteWeights(~G, S) (*procedure*), 149
Dense and Sparse Representations (*example*), 90
DepthFirstSearchTree(u) (*function*), 114, 167
Design(I, t) (*function*), 65
Design< t, P | X : parameters > (*constructor*), 49
Design< t, v | X : parameters > (*constructor*), 49
Developing a Difference Set (*example*), 56
Development(B) (*function*), 56
Development(T) (*function*), 56
DFSTree(u) (*function*), 114, 167
DiagonalSum(t1, t2) (*function*), 20
Diameter(G) (*function*), 112
DiameterPath(G) (*function*), 112
DifferenceSet(p, t) (*function*), 56
Digraph< n | edges : parameters> (*constructor*), 84
Digraph< S | edges : parameters> (*constructor*), 84
Distance(u, v : parameters) (*function*), 171
Distance(u, v) (*function*), 112
DistanceMatrix(G) (*function*), 113
DistancePartition(u) (*function*), 112
Distances(u : parameters) (*function*), 171
Dual(D) (*function*), 54

E

E ! < { u, v }, i > (*operation*), 144
E ! < [u, v], i > (*operation*), 144
E ! { u, v} (*operation*), 91
E ! [u, v] (*operation*), 91
E . i (*operation*), 91
e adj f (*operation*), 103, 161
e notadj f (*operation*), 103, 161
E.i {E . i} (*operation*), 144
Edge Decorations of a Multigraph (*example*), 150
EdgeCapacities(G) (*function*), 148
EdgeConnectivity(G : parameters) (*function*), 166
EdgeConnectivity(G) (*function*), 110
EdgeGroup(G) (*function*), 126
EdgeIndices(u, v) (*function*), 144
EdgeLabels(G) (*function*), 148
EdgeMultiplicity(u, v) (*function*), 144
Edges of a Multigraph (*example*), 145
Edges(G) (*function*), 91
Edges(u, v) (*function*), 144
EdgeSeparator(G : parameters) (*function*), 166
EdgeSeparator(G) (*function*), 110
EdgeSet(G) (*function*), 91
EdgeUnion(G, H) (*function*), 99, 159
EdgeUnion(N, H) (*function*), 159
EdgeWeights(G) (*function*), 148
ElementaryToHomogeneousMatrix(n) (*function*), 46
ElementaryToMonomialMatrix(n) (*function*), 45
ElementaryToPowerSumMatrix(n) (*function*), 46
ElementaryToSchurMatrix(n) (*function*), 45
ElementToSequence(w) (*function*), 7
Eltseq(w) (*function*), 7
Embedding(G) (*function*), 120, 168
Embedding(v) (*function*), 120, 168
EmptyDigraph(n : parameters) (*function*), 88
EmptyGraph(n : parameters) (*function*), 87
EndVertices(e) (*function*), 93, 144
EquitablePartition(P, G) (*function*), 112
EulerianGraphDatabase(n : parameters) (*function*), 134
EulerianNumber(n, r) (*function*), 2

F

Face(e) (*function*), 120, 168
Face(u, v) (*function*), 120, 168
Faces of a Planar Graph (*example*), 120
Faces(G) (*function*), 120, 168
Factorial(n) (*function*), 1
Fibonacci(n) (*function*), 1
Finitely Generated Ordered Monoid (*example*), 8
FirstIndexOfColumn(t, j) (*function*), 18
FirstIndexOfRow(t, i) (*function*), 18
Flow(e) (*function*), 186
Flow(u, v) (*function*), 186

for G in D do ... end for (*literal*), 133, 135
for random x in S do ... end for (*literal*), 93
for x in S do ... end for (*literal*), 93
Frobenius Automorphism (*example*), 39
Frobenius(s) (*function*), 39

G

G + { { u, v } } (*operation*), 96, 156
G + { u, v } (*operation*), 96, 155
G + { [u, v] } (*operation*), 96, 156
G + n (*operation*), 96, 155
G + [{ u, v }] (*operation*), 156
G + [u, v] (*operation*), 96, 155
G + [[u, v]] (*operation*), 156
G +:= { { u, v } } (*operation*), 96, 156
G +:= { u, v } (*operation*), 96, 156
G +:= { [u, v] } (*operation*), 96, 156
G +:= n (*operation*), 96, 155
G +:= [{ u, v }] (*operation*), 156
G +:= [u, v] (*operation*), 96, 156
G +:= [[u, v]] (*operation*), 156
G - { { u, v } } (*operation*), 97, 157
G - { e } (*operation*), 97, 157
G - { [u, v] } (*operation*), 97, 157
G - e (*operation*), 97, 157
G - U (*operation*), 96, 155
G - v (*operation*), 96, 155
G -:= { { u, v } } (*operation*), 97, 157
G -:= { e } (*operation*), 97, 157
G -:= { [u, v] } (*operation*), 97, 157
G -:= e (*operation*), 97, 157
G -:= U (*operation*), 96, 155
G -:= v (*operation*), 96, 155
G ^ n (*operation*), 100
G eq H (*operation*), 104, 162
G join H (*operation*), 99, 159
GeneralizedFibonacciNumber(g0, g1, n) (*function*), 1
GenerateGraphs(n : parameters) (*function*), 135
Geodesic(u, v : parameters) (*function*), 172
Geodesic(u, v) (*function*), 112
GeodesicExists(u, v : parameters) (*function*), 171
Geodesics(u : parameters) (*function*), 172
GewirtzGraph() (*function*), 102
Girth(G) (*function*), 113
GirthCycle(G) (*function*), 113
Graph Generation (*example*), 136
Graph(D, i) (*function*), 135
Graph(D, S, i) (*function*), 133
Graph< n | edges : parameters> (*constructor*), 82
Graph< S | edges : parameters > (*constructor*), 82
Graph< S | edges : parameters> (*constructor*), 82
Graphs(D, S) (*function*), 133
GraphSizeInBytes(n, m : parameters) (*function*), 81

H

Hadamard Designs (*example*), 75
Hadamard Equivalence Testing (*example*), 74
Hadamard Matrix Database (*example*), 76
HadamardAutomorphismGroup(H : parameters) (*function*), 75
HadamardCanonicalForm(H : parameters) (*function*), 73
HadamardColumnDesign(H, i) (*function*), 75
HadamardDatabase() (*function*), 76
HadamardDatabaseInformation(D : parameters) (*function*), 77
HadamardDatabaseInformationEmpty(: parameters) (*function*), 77
HadamardGraph(H : parameters) (*function*), 102
HadamardInvariant(H) (*function*), 73
HadamardMatrixFromInteger(x, n) (*function*), 74
HadamardMatrixToInteger(H) (*function*), 74
HadamardNormalize(H) (*function*), 73
HadamardRowDesign(H, i) (*function*), 75
HarmonicNumber(n) (*function*), 2
HasClique(G, k) (*function*), 116
HasClique(G, k, m : parameters) (*function*), 116
HasClique(G, k, m, f : parameters) (*function*), 116
HasDenseAndSparseRep(G) (*function*), 89
HasDenseRep(G) (*function*), 89
HasDenseRepOnly(G) (*function*), 89
HasElementaryBasis(A) (*function*), 35
HasHomogeneousBasis(A) (*function*), 35
HasMonomialBasis(A) (*function*), 35
HasNegativeWeightCycle(G) (*function*), 172
HasNegativeWeightCycle(u : parameters) (*function*), 172
HasParallelClass(D) (*function*), 63
HasParallelism(D: parameters) (*function*), 63
HasPowerSumBasis(A) (*function*), 35
HasResolution(D) (*function*), 62
HasResolution(D, λ) (*function*), 62
HasSchurBasis(A) (*function*), 35
HasSparseRep(G) (*function*), 89
HasSparseRepOnly(G) (*function*), 89
Homogeneous Functions as Sums of Monomial Functions (*example*), 34
Homogeneous-to-Elementary Transition Matrix (*example*), 44
Homogeneous-to-Schur Matrix as a Transpose (*example*), 44
HomogeneousToElementaryMatrix(n) (*function*), 44
HomogeneousToMonomialMatrix(n) (*function*), 44
HomogeneousToPowerSumMatrix(n) (*function*), 44
HomogeneousToSchurMatrix(n) (*function*), 44

HookLength(P, i, j) (*function*), 19
HookLength(t, i, j) (*function*), 19

I

Id(O) (*function*), 7
Id(P) (*function*), 9
Image(a, Y, y) (*function*), 129
Image(g, Y, y) (*function*), 68
IncidenceDigraph(A) (*function*), 85
IncidenceGraph(A) (*function*), 82
IncidenceGraph(D) (*function*), 70, 101
IncidenceGraph(P) (*function*), 102
IncidenceMatrix(D) (*function*), 58
IncidenceMatrix(G) (*function*), 113
IncidenceStructure(G) (*function*), 70
IncidenceStructure(I) (*function*), 65
IncidenceStructure< P | X > (*constructor*), 48
IncidenceStructure< v | X > (*constructor*), 48
IncidentEdges(u) (*function*), 93, 105, 106, 144, 163, 164
InDegree(u) (*function*), 105, 163
IndependenceNumber(G: parameters) (*function*), 118
Index(e) (*function*), 145
Index(v) (*function*), 91
IndexOfPartition(P) (*function*), 5
Indices(u, v) (*function*), 144
InitialVertex(e) (*function*), 93, 144
InNeighbors(u) (*function*), 106, 164
InNeighbours(u) (*function*), 106, 164
Inner Product of Symmetric Functions (*example*), 40
InnerProduct(a,b) (*function*), 40
InnerShape(t) (*function*), 17
InsertVertex(e) (*function*), 98, 158
InsertVertex(T) (*function*), 98, 158
IntersectionArray(G) (*function*), 132
IntersectionMatrix(G, P) (*function*), 113
IntersectionNumber(D, i, j) (*function*), 58
Invariants of a Fano Plane Design (*example*), 58
Invariants under Knuth Equivalence (*example*), 10
InverseJeuDeTaquin(~t, i, j) (*procedure*), 21
InverseJeuDeTaquin(t, i, j) (*function*), 21
InverseRowInsert(~t, i, j) (*procedure*), 21
InverseRowInsert(t, i, j) (*function*), 21
InverseRSKCorrespondenceDoubleWord(t1, t2) (*function*), 24
InverseRSKCorrespondenceMatrix(t1, t2) (*function*), 24
InverseRSKCorrespondenceSingleWord(t1, t2) (*function*), 23
IsBalanced(D, t: parameters) (*function*), 62
IsBiconnected(G) (*function*), 107, 164
IsBipartite(G) (*function*), 104, 162
IsBlock(D, S) (*function*), 60
IsBlockTransitive(D) (*function*), 69
IsCapacitated(E) (*function*), 148
IsCommutative(L) (*function*), 34
IsComplete(D) (*function*), 62
IsComplete(G) (*function*), 104, 162
IsConnected(G) (*function*), 106, 164
IsDesign(D, t: parameters) (*function*), 61
IsDifferenceSet(B) (*function*), 56
IsDirected(G) (*function*), 162
IsDistanceRegular(G) (*function*), 132
IsDistanceTransitive(G) (*function*), 132
IsDivisionRing(L) (*function*), 34
IsDomain(L) (*function*), 34
IsEdgeCapacitated(G) (*function*), 148
IsEdgeLabelled(G) (*function*), 148
IsEdgeTransitive(G) (*function*), 131
IsEdgeWeighted(G) (*function*), 148
IsEmpty(G) (*function*), 104, 162
IsEquitable(G, P) (*function*), 112
IsEuclideanDomain(L) (*function*), 34
IsEuclideanRing(L) (*function*), 34
IsEulerian(G) (*function*), 104
IsField(L) (*function*), 34
IsFinite(L) (*function*), 34
IsForest(G) (*function*), 104
IsHadamard(H) (*function*), 73
IsHadamardEquivalent(H, J : parameters) (*function*), 73
IsHomeomorphic(G : parameters) (*function*), 120
IsHomeomorphic(G: parameters) (*function*), 168
IsHomogeneous(s) (*function*), 37
IsIsomorphic(D, E: parameters) (*function*), 66
IsIsomorphic(G, H : parameters) (*function*), 126
IsIsomorphic(M, N: parameters) (*function*), 71
IsKEdgeConnected(G, k : parameters) (*function*), 166
IsKEdgeConnected(G, k) (*function*), 110
IsKnuthEquivalent(w1, w2) (*function*), 7
IsKVertexConnected(G, k : parameters) (*function*), 166
IsKVertexConnected(G, k) (*function*), 110
IsLabelled(E) (*function*), 147
IsLabelled(e) (*function*), 147
IsLabelled(u) (*function*), 146
IsLabelled(V) (*function*), 146
IsLexicographicallyOrdered(w1, w2) (*function*), 23
IsLinearSpace(D) (*function*), 61
IsLineRegular(D) (*function*), 62
IsLittlewoodRichardson(t) (*function*), 19
IsMinusOne(s) (*function*), 37
IsNearLinearSpace(D) (*function*), 61
IsNull(G) (*function*), 104, 162
Isomorphism of Coloured Graphs (*example*), 127
IsOne(s) (*function*), 37
IsOrdered(L) (*function*), 34

IsParallelClass(D, B, C) (*function*), 63
IsParallelism(D, P) (*function*), 63
IsPartition(S) (*function*), 4
IsPartitionRefined(G: parameters) (*function*), 125
IsPath(G) (*function*), 104
IsPID(L) (*function*), 34
IsPlanar(G) (*function*), 119, 167
IsPointRegular(D) (*function*), 62
IsPointTransitive(D) (*function*), 69
IsPolygon(G) (*function*), 104
IsPrimitive(G) (*function*), 131
IsPrincipalIdealRing(L) (*function*), 34
IsRegular(G) (*function*), 104, 162
IsResolution(D, P) (*function*), 63
IsReverseLatticeWord(w) (*function*), 7
IsRoot(v) (*function*), 114
IsRootedTree(G) (*function*), 114
IsSelfDual(D) (*function*), 61
IsSeparable(G) (*function*), 107, 164
IsSimple(D) (*function*), 61
IsSimple(G) (*function*), 162
IsSkew(t) (*function*), 19
IsStandard(t) (*function*), 19
IsSteiner(D, t) (*function*), 62
IsStronglyConnected(G) (*function*), 107, 165
IsSubgraph(G, H) (*function*), 104, 162
IsSymmetric(D) (*function*), 62
IsSymmetric(G) (*function*), 131
IsTransitive(G) (*function*), 131
IsTree(G) (*function*), 104
IsTriconnected(G) (*function*), 108, 165
IsTrivial(D) (*function*), 61
IsUFD(L) (*function*), 34
IsUndirected(G) (*function*), 162
IsUniform(D) (*function*), 61
IsUnitary(L) (*function*), 34
IsVertexLabelled(G) (*function*), 146
IsVertexTransitive(G) (*function*), 131
IsWeaklyConnected(G) (*function*), 107, 165
IsWeighted(E) (*function*), 148
IsZero(s) (*function*), 37

J

Jeu de Taquin (*example*), 21
JeuDeTaquin(~t) (*procedure*), 20
JeuDeTaquin(~t, i, j) (*procedure*), 20
JeuDeTaquin(t) (*function*), 20
JeuDeTaquin(t, i, j) (*function*), 20

K

KCubeGraph(n : parameters) (*function*), 87
Knuth Equivalence of Row and Column Words (*example*), 20
Kostka Numbers (*example*), 42
KostkaNumber(S, C) (*function*), 26

L

L eq M (*operation*), 34
L ne M (*operation*), 34
Label(e) (*function*), 148
Label(u) (*function*), 146
Labels(E) (*function*), 148
Labels(S) (*function*), 146, 148
Labels(V) (*function*), 146
LastIndexOfColumn(t, j) (*function*), 18
LastIndexOfRow(t, i) (*function*), 18
Length(s) (*function*), 38
Length(u) (*function*), 9
Length(w) (*function*), 7
LexicographicalOrdering(~w1, ~w2) (*procedure*), 23
LexicographicalOrdering(w1, w2) (*function*), 23
LexProduct(G, H) (*function*), 100
Line(D, p, q) (*function*), 60
LinearCode(D, K) (*function*), 70
LinearSpace(I) (*function*), 65
LinearSpace< P | X : parameters > (*constructor*), 49
LinearSpace< v | X : parameters > (*constructor*), 49
LineGraph(G) (*function*), 98
LineGraph(P) (*function*), 102
Longest Increasing Subsequences (*example*), 8
Lucas(n) (*function*), 1

M

M ! Q (*operation*), 13
M ! u (*operation*), 13
M ! w (*operation*), 13
M ! [i_1, ..., i_n] (*operation*), 13
M ^ x (*operation*), 71
Mapping Vertices of a Subgraph (*example*), 94
Maps between Ordered, Plactic, and Tableau Monoids (*example*), 14
Matrices(D, n) (*function*), 76
Matrix(D, n, k) (*function*), 76
Maxdeg(G) (*function*), 105, 106, 163
MaximalIncreasingSequence(w) (*function*), 7
MaximalIncreasingSequences(w, k) (*function*), 7
Maximum Matching in a Bipartite Graph (*example*), 109
MaximumClique(G : parameters) (*function*), 116
MaximumDegree(G) (*function*), 105, 106, 163
MaximumFlow(s, t : parameters) (*function*), 185
MaximumFlow(Ss, Ts : parameters) (*function*), 185
MaximumInDegree(G) (*function*), 106, 163
MaximumIndependentSet(G: parameters) (*function*), 117
MaximumMatching(G : parameters) (*function*), 165

MaximumMatching(G) (*function*), 109
MaximumOutDegree(G) (*function*), 106, 163
Maxindeg(G) (*function*), 106, 163
Maxoutdeg(G) (*function*), 106, 163
Mindeg(G) (*function*), 105, 106, 163, 164
Minimum Weight Tree of a Multigraph (*example*), 173
MinimumCut(s, t : parameters) (*function*), 185
MinimumCut(Ss, Ts : parameters) (*function*), 185
MinimumDegree(G) (*function*), 105, 106, 163, 164
MinimumDominatingSet(G) (*function*), 105
MinimumInDegree(G) (*function*), 106, 163
MinimumOutDegree(G) (*function*), 106, 163
MinimumWeightTree(u : parameters) (*function*), 172
Minindeg(G) (*function*), 106
Minindeg(G)) (*function*), 163
Minoutdeg(G) (*function*), 106, 163
MonomialToElementaryMatrix(n) (*function*), 43
MonomialToHomogeneousMatrix(n) (*function*), 43
MonomialToPowerSumMatrix(n) (*function*), 43
MonomialToSchurMatrix(n) (*function*), 43
MultiDigraph<n | edges > (*constructor*), 142
MultiDigraph<S | edges > (*constructor*), 142
MultiGraph<n | edges > (*constructor*), 141
MultiGraph<S | edges > (*constructor*), 141
Multinomial(n, [r_1, ... r_n]) (*function*), 1
MultipartiteGraph(Q) (*function*), 87
Multiplication of Symmetric Functions (*example*), 37
Multiplicity(u, v) (*function*), 144
Multisets(S, k) (*function*), 2

N

N + < [u, v], c > (*operation*), 182
N + { < [u, v], c > } (*operation*), 182
N + [< [u, v], c >] (*operation*), 182
N +:= < [u, v], c > (*operation*), 182
N +:= { < [u, v], c > } (*operation*), 182
N +:= [< [u, v], c >] (*operation*), 182
N join H (*operation*), 159
NearLinearSpace(I) (*function*), 65
NearLinearSpace< P | X : parameters > (*constructor*), 48
NearLinearSpace< v | X : parameters > (*constructor*), 48
Neighbors(u) (*function*), 105, 163
Neighbours(u) (*function*), 105, 163
Network from a Digraph (*example*), 176
Network from Capacity Tuples (*example*), 177
Network<n | edges > (*constructor*), 175
Network<S | edges > (*constructor*), 175
NextGraph(I: parameters) (*function*), 136
NFaces(G) (*function*), 120, 168
NullGraph(: parameters) (*function*), 87
NumberOfBlocks(D) (*function*), 57
NumberOfClasses(D) (*function*), 133
NumberOfEdges(G) (*function*), 102, 161
NumberOfFaces(G) (*function*), 120, 168
NumberOfGraphs(D) (*function*), 133
NumberOfGraphs(D, S) (*function*), 133
NumberOfMatrices(D, n) (*function*), 76
NumberOfPartitions(n) (*function*), 4
NumberOfPermutations(n, k) (*function*), 1
NumberOfPoints(D) (*function*), 57
NumberOfRows(t) (*function*), 17
NumberOfSkewRows(t) (*function*), 17
NumberOfStandardTableaux(P) (*function*), 26
NumberOfStandardTableauxOnWeight(n) (*function*), 26
NumberOfTableauxOnAlphabet(P, m) (*function*), 26
NumberOfVertices(G) (*function*), 102, 161

O

O ! 1 (*operation*), 7
O ! [i_1, ..., i_n] (*operation*), 7
O ! [w_1, ..., w_n] (*operation*), 7
O . i (*operation*), 7
Obstruction(G) (*function*), 120, 167
OddGraph(n) (*function*), 102
OpenGraphFile(s, f, p) (*function*), 137
Operations on Points and Blocks (*example*), 60
OptimalEdgeColouring(G) (*function*), 115
OptimalVertexColouring(G) (*function*), 114
Orbit(A, Y, y) (*function*), 129
Orbit(G, Y, y) (*function*), 68
OrbitalGraph(P, u, T) (*function*), 101
Orbits(A, Y) (*function*), 129
Orbits(G, Y) (*function*), 68
OrbitsPartition(G) (*function*), 131
Order(D) (*function*), 58
Order(G) (*function*), 102, 161
Ordered and Plactic Integer Monoids (*example*), 9
Ordered Monoid over the Integers (*example*), 8
OrderedIntegerMonoid() (*function*), 7
OrderedMonoid(M) (*function*), 11
OrderedMonoid(n) (*function*), 7
OrderedMonoid(P) (*function*), 9
OrientatedGraph(G) (*function*), 100, 159
OutDegree(u) (*function*), 105, 163
OuterShape(t) (*function*), 17
OutNeighbors(u) (*function*), 106, 164
OutNeighbours(u) (*function*), 106, 164

P

P ! 1 (*operation*), 9
P ! s (*operation*), 38
P ! t (*operation*), 9
P ! w (*operation*), 9
P ! x (*operation*), 52
P ! [i_1, ..., i_n] (*operation*), 9

P ! [u_1, ..., u_n] (*operation*), 9
P ! [w_1, ..., w_n] (*operation*), 9
P . i (*operation*), 9, 52
p in B (*operation*), 59
p notin B (*operation*), 59
PaleyGraph(q) (*function*), 101
PaleyTournament(q) (*function*), 101
Parameters(D) (*function*), 58
Parent(f) (*function*), 35
Parent(L) (*function*), 34
ParentGraph(S) (*function*), 93
ParentGraph(s) (*function*), 93
PartitionCovers(P1, P2) (*function*), 17
Partitions and Conjugacy Classes (*example*), 5
Partitions(n) (*function*), 4
Partitions(n, k) (*function*), 4
PascalTriangle(D) (*function*), 58
Path(u, v : parameters) (*function*), 171
PathExists(u, v : parameters) (*function*), 171
PathGraph(n : parameters) (*function*), 87
Paths(u : parameters) (*function*), 171
Permutations(S) (*function*), 2
Permutations(S, k) (*function*), 2
Petersen Graph as an Odd Graph (*example*), 2
PetersenGraph() (*function*), 102
PlacticIntegerMonoid() (*function*), 9
PlacticMonoid(O) (*function*), 9
Planar Dual and Minimal Cuts (*example*), 169
Planar Embedding of a Multigraph (*example*), 168
PlanarDual(G) (*function*), 120
PlanarGraphDatabase(n) (*function*), 135
Plethysm of Symmetric Functions (*example*), 37
Point(D, i) (*function*), 52
PointDegree(D, p) (*function*), 60
PointDegrees(D) (*function*), 57
PointGraph(D) (*function*), 70, 101
PointGraph(P) (*function*), 102
PointGroup(D) (*function*), 67
Points and Blocks of an Incidence Structure (*example*), 53
Points(D) (*function*), 57
PointSet(D) (*function*), 52
PolygonGraph(n : parameters) (*function*), 87
PowerSumToElementaryMatrix(n) (*function*), 45
PowerSumToHomogeneousMatrix(n) (*function*), 45
PowerSumToMonomialMatrix(n) (*function*), 45
PowerSumToSchurMatrix(n) (*function*), 44
pRank(D, p) (*function*), 58
PrimeRing(L) (*function*), 34
Properties of a Skew Tableau (*example*), 18

Q

quo< G | P > (*constructor*), 94
Quotient of a Complete Graph (*example*), 95

R

Random Tableaux (*example*), 16
Random(B) (*function*), 53
Random(b) (*function*), 53
Random(D) (*function*), 135
Random(P) (*function*), 52
Random(S) (*function*), 93
RandomDigraph(n, r : parameters) (*function*), 88
RandomGraph(D) (*function*), 133
RandomGraph(D, S) (*function*), 133
RandomGraph(n, r : parameters) (*function*), 87
RandomHookWalk(P, i, j) (*function*), 16
RandomHookWalk(t, i, j) (*function*), 16
RandomPartition(n) (*function*), 5
RandomTableau(M, n) (*function*), 16
RandomTableau(M, S) (*function*), 16
RandomTableau(n) (*function*), 16
RandomTableau(S) (*function*), 16
RandomTree(n : parameters) (*function*), 88
Reachable(u, v : parameters) (*function*), 171
Reachable(u, v) (*function*), 112
Reading a Graph File (*example*), 138
Rectify(~t) (*procedure*), 20
Rectify(t) (*function*), 20
Redundant Terms after Coercion (*example*), 32
Related Structures of a Design (*example*), 54
RemoveEdge(~G, e) (*procedure*), 97, 157
RemoveEdge(~G, u, v) (*procedure*), 97, 157
RemoveEdges(~G, S) (*procedure*), 97, 157
RemoveVertex(~G, v) (*procedure*), 96, 155
RemoveVertices(~G, U) (*procedure*), 96, 155
Rep(B) (*function*), 52
Rep(b) (*function*), 53
Rep(P) (*function*), 52
Rep(S) (*function*), 93
ReplicationNumber(D) (*function*), 58
Representative(B) (*function*), 52
Representative(b) (*function*), 53
Representative(P) (*function*), 52
Representative(S) (*function*), 93
Residual(D, b) (*function*), 54
Residual(D, p) (*function*), 54
Resolutions and Parallel Classes (*example*), 63
RestrictDegree(a, n) (*function*), 41
RestrictedPartitions(n, k, M) (*function*), 4
RestrictedPartitions(n, M) (*function*), 4
Restricting to Finitely Many Variables (*example*), 38
Restriction(D, S) (*function*), 54
RestrictPartitionLength(a, n) (*function*), 41
RestrictParts(a, n) (*function*), 41
Root(G) (*function*), 114
RootSide(v) (*function*), 114
Row(t, i) (*function*), 17
RowInsert(~t, u) (*procedure*), 21
RowInsert(~t, w) (*procedure*), 21

RowInsert(~t, x) (*procedure*), 21
RowInsert(t, u) (*function*), 21
RowInsert(t, w) (*function*), 21
RowInsert(t, x) (*function*), 21
RowLength(t, i) (*function*), 18
Rows(t) (*function*), 17
RowSkewLength(t, i) (*function*), 18
RowWord(t) (*function*), 19
RSK Correspondence and Inverse Permutations (*example*), 26
RSK Correspondence for a Single Word (*example*), 24
RSK Correspondence for Matrices (*example*), 25
RSK Correspondence for Two Words (*example*), 25
RSKCorrespondence(M) (*function*), 24
RSKCorrespondence(w) (*function*), 23
RSKCorrespondence(w1, w2) (*function*), 23

S

S eq T (*operation*), 92
s eq t (*operation*), 37, 92, 145
s in S (*operation*), 92
S ne T (*operation*), 92
s ne t (*operation*), 37, 93
s notin S (*operation*), 92
S notsubset B (*operation*), 60
S notsubset T (*operation*), 92
S subset B (*operation*), 60
S subset T (*operation*), 92
Scalar Symmetric Functions (*example*), 33
SchreierGraph(A, B) (*function*), 101
SchreierMultiGraph(A, B : parameter) (*function*), 153
SchurToElementaryMatrix(n) (*function*), 42
SchurToHomogeneousMatrix(n) (*function*), 42
SchurToMonomialMatrix(n) (*function*), 42
SchurToPowerSumMatrix(n) (*function*), 42
SelfComplementaryGraphDatabase(n) (*function*), 135
SeparationVertices(G) (*function*), 108, 165
Set(B) (*function*), 60
Setting the Print Style (*example*), 35
SetVerbose("HadamardDB", v) (*procedure*), 78
SFA(R) (*function*), 30
SFAElementary(R) (*function*), 30
SFAHomogeneous(R) (*function*), 30
SFAMonomial(R) (*function*), 31
SFAPower(R) (*function*), 30
SFASchur(R) (*function*), 30
Shape(t) (*function*), 17
Shortest Paths in a Multidigraph (*example*), 172
ShortestPath(u, v : parameters) (*function*), 171
ShortestPaths(u : parameters) (*function*), 171
ShrikhandeGraph() (*function*), 102
Simplify(D) (*function*), 54
SingerDifferenceSet(n, q) (*function*), 56
Size Limit of a Dense Graph (*example*), 81
Size(G) (*function*), 102, 161
Skew Tableau over Finitely Many Labels (*example*), 13
SkewHadamardDatabase() (*function*), 76
SkewShape(t) (*function*), 17
SkewWeight(t) (*function*), 17
SmallGraphDatabase(n : parameters) (*function*), 134
SpanningForest(G) (*function*), 113, 167
SpanningTree(G) (*function*), 113, 167
Spectrum(G) (*function*), 103
Sphere(u, n) (*function*), 112
Splitcomponents(G) (*function*), 108, 165
SquareLatticeGraph(n) (*function*), 102
Stabilizer(A, Y, y) (*function*), 129
Stabilizer(G, Y, y) (*function*), 69
Standard Graph of an Odd Graph (*example*), 86
Standard Tableaux of a Given Shape (*example*), 15
StandardGraph(G) (*function*), 86, 143
StandardTableaux(M, P) (*function*), 14
StandardTableaux(P) (*function*), 14
StandardTableauxOfWeight(M, n) (*function*), 14
StandardTableauxOfWeight(n) (*function*), 14
StirlingFirst(n, k) (*function*), 1
StirlingSecond(n, k) (*function*), 1
StronglyConnectedComponents(G) (*function*), 107, 165
StronglyRegularGraphsDatabase() (*function*), 133
sub< G | list > (*constructor*), 94, 151
sub< N | list > (*constructor*), 178
Subgraphs of a Decorated Multidigraph (*example*), 152
Subgraphs of a Network (*example*), 179
Subsequences(S, k) (*function*), 2
Subsets(S) (*function*), 2
Subsets(S, k) (*function*), 2
Sum(Q) (*function*), 54
Support of a Multigraph (*example*), 143
Support(B) (*function*), 60
Support(D) (*function*), 57
Support(G) (*function*), 85, 143
Support(s) (*function*), 37
Support(V) (*function*), 85, 143
Switch(S) (*function*), 98
Switch(u) (*function*), 98
Symmetric Function Algebras with Different Bases (*example*), 31
Symmetric Functions as Linear Combinations (*example*), 31
Symmetric Functions in Any Number of Variables (*example*), 29
SymmetricCharacter(sf) (*function*), 40
SymmetricFunctionAlgebra(R) (*function*), 30

SymmetricFunctionAlgebraElementary(R) (*function*), 30
SymmetricFunctionAlgebraHomogeneous(R) (*function*), 30
SymmetricFunctionAlgebraMonomial(R) (*function*), 31
SymmetricFunctionAlgebraPower(R) (*function*), 30
SymmetricFunctionAlgebraSchur(R) (*function*), 30
Symmetry of the 8-Cube Graph (*example*), 132
Symmetry of the Elementary-to-Monomial Matrix (*example*), 46
Symmetry of the Homogeneous-to-Monomial Matrix (*example*), 44

T

t1 * t2 (*operation*), 20
t1 eq t2 (*operation*), 20
Tableau Monoid over Finitely Many Labels (*example*), 12
Tableau Monoid over the Integers (*example*), 11
Tableau(Q) (*function*), 12
Tableau(S, Q) (*function*), 12
TableauIntegerMonoid() (*function*), 11
TableauMonoid(O) (*function*), 11
TableauMonoid(P) (*function*), 11
Tableaux with a Given Shape and Content (*example*), 15
Tableaux(sf, m) (*function*), 40
TableauxOfShape(M, S, m) (*function*), 15
TableauxOfShape(S, m) (*function*), 15
TableauxOnShapeWithContent(M, S, C) (*function*), 15
TableauxOnShapeWithContent(S, C) (*function*), 15
TableauxWithContent(C) (*function*), 15
TableauxWithContent(M, C) (*function*), 15
TensorProduct(G, H) (*function*), 100
TerminalVertex(e) (*function*), 93, 144
Transition Matrices Agree with Coercion (*example*), 42
TriangularGraph(n) (*function*), 102
Triconnectivity of a Graph (*example*), 108
Tutte's 8-Cage (*example*), 83
Two Ways of Multiplying Tableaux (*example*), 22
Two-Colouring a Complete Bipartite Graph (*example*), 149

U

u adj v (*operation*), 103, 161
u in e (*operation*), 103, 161
u notadj v (*operation*), 103, 161
u notin e (*operation*), 103, 161
u1 * u2 (*operation*), 9
u1 eq u2 (*operation*), 9
UncapacitatedGraph(G) (*function*), 149
UnderlyingDigraph(G) (*function*), 100, 160
UnderlyingGraph(D) (*function*), 100
UnderlyingGraph(G) (*function*), 160
UnderlyingMultiDigraph(G) (*function*), 160
UnderlyingMultiGraph(G) (*function*), 160
UnderlyingNetwork(G) (*function*), 160
Union(D, E) (*function*), 54
Union(G, H) (*function*), 99, 159
Union(N, H) (*function*), 159
UnlabelledGraph(G) (*function*), 149
UnweightedGraph(G) (*function*), 149
UpdateHadamardDatabase(~R, S : parameters) (*procedure*), 77
Updating the Hadamard Database (*example*), 78

V

V ! v (*operation*), 91
V . i (*operation*), 91
Valence(G) (*function*), 105
Vertex and Edge Connectivity (*example*), 110
VertexConnectivity(G : parameters) (*function*), 166
VertexConnectivity(G) (*function*), 110
VertexLabels(G) (*function*), 146
VertexPath(u,v) (*function*), 114
VertexSeparator(G : parameters) (*function*), 165
VertexSeparator(G) (*function*), 109
VertexSet(G) (*function*), 91
Vertices and Edges of an Odd Graph (*example*), 91
Vertices(G) (*function*), 91

W

w1 * w2 (*operation*), 7
w1 eq w2 (*operation*), 7
Weight(e) (*function*), 148
Weight(P) (*function*), 5
Weight(t) (*function*), 17
WeightedPartitions(n, w) (*function*), 4
WeightedRestrictedPartitions(n, S, w) (*function*), 4
Weights(E) (*function*), 148
Weights(S) (*function*), 148
WittDesign(n) (*function*), 56
Word(t) (*function*), 19
WordToTableau(u) (*function*), 12
WordToTableau(w) (*function*), 12
WriteHadamardDatabase(S, ~R) (*procedure*), 77
WriteRawHadamardData(S, R) (*procedure*), 78
w[i] (*indexed*), 7