The COCO2P Package

Version 0.16

Mikhail Klin
Christian Pech
Sven Reichard

Mikhail Klin Email: klin@math.bgu.ac.il

Christian Pech Email: christian.pech@tu-dresden.de

Sven Reichard Email: sven.reichard@tu-dresden.de

Copyright

Contents

Chapter 1

Color Graphs

1.1 Theory

A color graph (cgr) in COCO2P is a triple (V, C, f), where V is a set of vertices, C is set of colors, and $f : V \times V \to C$ assigns to every arc its color. COCO2P does not know the concept of non-arcs. However, this is not an essential restriction, since non-arcs may be simulated by introducing a special distinguished color.

Of special interrest in COCO2P are WL-stable color graphs (that is cgrs that are stable under the Weisfeiler-Leman algorithm [WL68], [Wei76]). In the frames of COCO2P the maximal monochromatic sets of arcs of a WL-stable color graph will always form a *coherent configuration*. On the other hand, from every coherent configuration we can obtain a WL-stable cgr (every pair of vertices is colored by the relation it belongs to).

For historical reasons, COCO2P uses the nomenclature of color graphs. However, this is only of importance for concepts like automorphisms and isomorphisms. While both, automorphisms and isomorphisms have to preserve colors (as it is expected for color graphs), color-automorphisms, and color-isomorphisms only have to respect color classes, that is, they may map arcs of one color to arcs of another color.

1.2 On the representation of color graphs in COCO2P

For a color graph, the set of vertices as well as the set of colors may be any finite set representable in GAP. For performance reasons, COCO2P does not use these sets inside its algorithms (except when constructing color graphs). Instead, COCO2P refers to vertices and colors by their position in the vertex-set and color-set, respectively. In fact vertices and colors are identified with these indices. In order not to loose information, every color graph in COCO2P keeps a list of names of vertices and a list of names of colors. The set of vertex-names is equal to the original set of vertices, and the set of color names is equal to the original set of colors.

1.3 Functions for the construction of color graphs

1.3.1 ColorGraphByOrbitals

▷ `ColorGraphByOrbitals(`*grp[, domain[, Action[, completeDom]]]*`)` (function)

This function constructs the color graph of orbitals color graphs from a group action by mapping each arc to a representative of its orbital of the given group action.

In its first form, the function returns the color graph of orbitals of the permutation group *grp* in its natural action (i.e. on $\{1,\ldots,n\}$, where n is the largest moved point of *grp*).

Example
```
gap> d7 := Group( (1,2,3,4,5,6,7), (1,7)(2,6)(3,5));;
gap> cgr := ColorGraphByOrbitals(d7);
<color graph of order 7 and rank 4>
```

In the second form the function returns the color graph of orbitals of *grp* acting on *domain* OnPoints. If *domain* is not invariant under *grp*, then the smallest invariant extension of *domain* is taken as acting domain.

Example
```
gap> d7 := Group( (1,2,3,4,5,6,7), (1,7)(2,6)(3,5));;
gap> cgr := ColorGraph(d7, [1]);
<color graph of order 7 and rank 4>
```

In the third variant of ColorGraphByOrbitals an action can be given:

Example
```
gap> cgr:=ColorGraph(SymmetricGroup(5), Combinations([1..5],2), OnSets);
<color graph of order 10 and rank 3>
```

The optional fourth argument *completeDom* is a boolean. If it is True, then the function assumes that *domain* is closed under *action* of *grp*. This has the effect, that the function dows not try to complete it. The effect is that in the resulting color graph it is guaranteed that the vertex with number i corresponds exactly to domain[i].

1.3.2 ColorGraph

▷ ColorGraph(*grp[, domain[, action[, completeDom[, coloring]]]]*) (function)

This is the most general function for the construction of color graphs. When called with less than 5 arguments, it is identical with the function ColorGraphByOrbitals (1.3.1)

The optional fifth argument *coloring* is a coloring-function. It takes as input two vertices (elements of the acting domain) u,v, and it has to return the color of the arc (u,v). In principle, the color can be any GAP object. However, it should be possible to compare colors and to form sets of them.

Example
```
gap> cgr:=ColorGraph(SymmetricGroup(8),
> Combinations([1..8],4), OnSets, true,
> function(u,v) return
> Length(Intersection(u,v));end);
<color graph of order 70 and rank 5>
```

It is supposed that *coloring* is invariant under the given action (this is not checked!).

1.3.3 ColorGraphByMatrix

▷ `ColorGraphByMatrix(mat)` (function)

This function constructs a color graph from its adjacency matrix. The argument *mat* is a list of n lists of length n. The vertex-set of the resulting color graph is $\{1,\dots,n\}$, while the color of the arc (i,j) is `mat[i][j]`. The entries can be any kind of GAP-objects that can be compared and that can be organized in a set.

Example

```
gap> m:=[["black","red"  ,"blue" ,"blue" ,"blue" ],
>        ["blue" ,"black","red"  ,"blue" ,"blue" ],
>        ["blue" ,"blue", "black","red"  ,"blue" ],
>        ["blue" ,"blue", "blue" ,"black","red"  ],
>        ["red"  ,"blue", "blue" ,"blue" ,"black"]];;
gap> cgr:=ColorGraphByMatrix(m);
<color graph of order 5 and rank 3>
```

1.3.4 ColorGraphByWLStabilization

▷ `ColorGraphByWLStabilization(cgr)` (function)

If *cgr* is WL-stable then the function returns *cgr*. Otherwise, the WL-stabilization of *cgr* is returned. The colors of the stabilization have names of the shape `[c,i]` where `c` is a color of *cgr* and `i` is the index of a fragment of color `c`.

This function does not really implement the Weisfeiler-Leman algorithm. Rather it does a stabilization inside of a Schurian WL-stable fission of *cgr*. The performance depends mainly on the order of the group of known automorphisms of *cgr* (cf `KnownGroupOfAutomorphisms` (1.7.1)).

1.3.5 ClassicalCompleteAffineScheme

▷ `ClassicalCompleteAffineScheme(q)` (function)

The classical complete affine scheme is a WL-stable, Schurian, amorphic color graph defined on the set of points of the affine plane over $GF(q)$. The reflexive closure of every irreflexive color class is an equivalence relation whose equivalence classes form a complete parallel class of lines. Moreover, to every parallel class there corresponds a color class.

This function returns the classical complete affine scheme over $GF(q)$.

1.3.6 JohnsonScheme

▷ `JohnsonScheme(n, k)` (function)

The Johnson scheme $J(n,k)$ is a WL-stable, Schurian color graph. Its vertices are the k-element subsets of $\{1,\dots,n\}$. The colors are elements of $\{0,\dots,k\}$. The color of an arc (M,N) is the cardinality of the intersection of M and N.

This function returns the Johnson scheme $J(n,k)$.

1.3.7 CyclotomicColorGraph

▷ `CyclotomicColorGraph(p, n, d)` (function)

Let p be a prime, n, d be positive integers, such that d divides $(p^n - 1)$. Let $q := p^n$, and let r be a primitive element of $GF(q)$. Let C be the set of all powers of r^d in $GF(q)$ the cyclotomic colored graph $Cyc(p,n,d)$ has as vertices the elements of $GF(q)$. The set of colors is given by $\{*,0,1,...,d-1\}$. A pair (x,y) of vertices has color $*$ in $Cyc(p,n,d)$ if $x = y$. It has color i if $(x-y)$ is an element of $C\cdot(r^i)$.

This function returns the Cyclotomic scheme $Cyc(p,n,d)$.

1.3.8 BIKColorGraph

▷ `BIKColorGraph(m)` (function)

This function generates the color graphs described in the paper [BIK89]. These color graphs are interesting because they may be used to construct 3-isoregular strongly regular graphs with the 5-vertex condition. The vertex set of `BIKColorGraph(m)` is $V = GF(2)^{2m}$. For the description of colors of the arcs consider a quadratic form q of Witt-index m on V. Let Q be the quadric defined by q, and let S be a maximal singular subspace of q. A pair of vectors (v,w) is colored by

`"="`:
if $v = w$,

`"Q+S+"`:
if $v + w \in S$,

`"Q+S-"`:
if $v + w \in Q \setminus S$,

`"Q-"`:
if $v + w \notin Q$.

The following code constructs the Ivanov-graph on 256 vertices. This was historically the first strongly regular graph to be found that is non-rank-3 and that satisfies the 5-vertex condition (cf. [Iva89]).

Example
```
gap> cgr:=BIKColorGraph(4);
    <color graph of order 256 and rank 4>
    gap> ColorNames(cgr);
    [ "=", "Q+S+", "Q+S-", "Q-" ]
    gap> gamma:=BaseGraphOfColorGraph(cgr,3);;
    gap> IsStronglyRegular(gamma);
    true
    gap> gamma.srg;
    rec( k := 120, lambda := 56, mu := 56, r := 8, s := -8, v := 256 )
```

1.3.9 IvanovColorGraph

▷ `IvanovColorGraph(m)` (function)

This function generates a series of color graphs described in [Iva94]. These color graphs may be used to construct 3-isoregular strongly regular graphs with the 5-vertex condition. These color graphs are interesting because they may be used to construct 3-isoregular strongly regular graphs with the 5-vertex condition. The vertex set of `IvanovColorGraph(m)` is $V = GF(2)^{2m}$. For the description of colors of the arcs consider a quadratic form q of Witt-index $m-1$ on V. Let Q be the quadric defined by q, let S be a maximal singular subspace of q, and let O be the orthogonal complement of S. A pair of vectors (v,w) is colored by

`"="`:
: if $v = w$,

`"Q+S+"`:
: if $v + w \in S$,

`"Q+S-"`:
: if $v + w \in Q \setminus S$,

`"Q-O+"`:
: if $v + w \in O$.

`"Q-O-"`:
: if $v + w \notin O \cup Q$.

Example

```
gap> cgr:=IvanovColorGraph(5);
    <color graph of order 1024 and rank 5>
    gap> ColorNames(cgr);
    [ "=", "Q+S+", "Q+S-", "Q-O+", "Q-O-" ]
    gap> gamma:=BaseGraphOfColorGraph(cgr,[2,5]);;
    gap> IsStronglyRegular(gamma);;
    gap> gamma.srg;
    rec( k := 495, lambda := 238, mu := 240, r := 15, s := -17, v := 1024 )
```

1.3.10 AllAssociationSchemes

▷ `AllAssociationSchemes(n)` (function)

This function creates an interface to the database of small association schemes by Akihide Hanaki and Izumi Miyamoto from `http://math.shinshu-u.ac.jp/~hanaki/as/` (further refered to as the Japanese catalogue)

This function downloads the list of small association schemes of order *n*. Then it converts them to the internal format of COCO2P and returns the resulting list. Every color graph has a name of the shape `AS(n,k)` where `k` is the index of the scheme in the list of schemes of order *n* in the Japanese catalogue.

1.3.11 AllCoherentConfigurations

▷ `AllCoherentConfigurations(n)` (function)

This function creates an interface to the database of small coherent configurations on at most 15 vertices by Matan Ziv-Av This function downloads the list of small coherent configurations of order *n*. Then it converts them to the internal format of COCO2P and returns the resulting list. Every color graph has a name of the shape `CC(n,k)` where `k` is the index of the scheme in the list of schemes of order *n* in Matan's catalogue.

1.4 Functions for the inspection of color graphs

1.4.1 OrderOfColorGraph

▷ `OrderOfColorGraph(cgr)` (attribute)
▷ `OrderOfCocoObject(cgr)` (attribute)
▷ `Order(cgr)` (attribute)

Returns the number of vertices of *cgr*.

1.4.2 RankOfColorGraph

▷ `RankOfColorGraph(cgr)` (attribute)
▷ `Rank(cgr)` (method)

Returns the number of colors of *cgr*.

1.4.3 VertexNamesOfCocoObject (for color graphs)

▷ `VertexNamesOfCocoObject(cgr)` (operation)
▷ `VertexNamesOfColorGraph(cgr)` (operation)

Returns the list of names of the vertices of *cgr*. Unfortunately, the more elegant name `VertexNames` is used in Grape as the name of a global function and can not be overloaded.

Example
```
gap> cgr:=JohnsonScheme(5,2);;
gap> VertexNamesOfCocoObject(cgr);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ], [ 2, 4 ], [ 2, 5 ],
[ 3, 4 ], [ 3, 5 ], [ 4, 5 ] ]
```

1.4.4 ColorNames

▷ `ColorNames(cgr)` (operation)

Returns the list of names of the colors of *cgr*. In the following example, the color names of the Johnson scheme are the possible cardinalities of the intersection of two 2-element subsets of $\{1,2,3,4,5\}$. Thus loobs will get colored by 1, since the intersection of a 2-element set with itself will have cardinality 2.

Example
```
gap> cgr:=JohnsonScheme(5,2);;
gap> ColorNames(cgr);
```

```
[ 2, 1, 0 ]
```

1.4.5 ArcColorOfColorGraph (first variant)

▷ ArcColorOfColorGraph(*cgr*, *u*, *v*) (method)
▷ ArcColorOfColorGraph(*cgr*, *arc*) (method)

Returns the color of the arc (u,v). In the second form, the arc is *arc* is given as an ordered pair [u,v].

Example

```
gap> cgr:=JohnsonScheme(5,2);;
gap> ColorNames(cgr);
[ 2, 1, 0 ]
gap> VertexNamesOfCocoObject(cgr);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ], [ 2, 4 ], [ 2, 5 ],
[ 3, 4 ], [ 3, 5 ], [ 4, 5 ] ]
gap> ArcColorOfColorGraph(cgr,1,10);
3
gap> ArcColorOfColorGraph(cgr,[2,9]);
2
```

1.4.6 ColorRepresentative

▷ ColorRepresentative(*cgr*, *i*) (operation)

Returns any arc of color *i* of *cgr*.

1.4.7 Neighbors (first variant)

▷ Neighbors(*cgr*, *vertices*, *colors*) (method)
▷ Neighbors(*cgr*, *v*, *colors*) (method)
▷ Neighbors(*cgr*, *vertices*, *color*) (method)
▷ Neighbors(*cgr*, *v*, *color*) (method)

The first variant returns the set of all vertices w of *cgr* such that the color of the arc (v,w) is an element of the set *colors*, for all v in *vertices*.

The second variant gets as the second argument a single vertex of *cgr*, the third gets a single color and the fourth variant gets both, a single vertex and a single color..

Example

```
gap> cgr:=JohnsonScheme(5,2);;
gap> ColorNames(cgr);
[ 2, 1, 0 ]
gap> VertexNamesOfCocoObject(cgr);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ], [ 2, 4 ], [ 2, 5 ],
[ 3, 4 ], [ 3, 5 ], [ 4, 5 ] ]
gap> Neighbors(cgr,1,3);
[ 8, 9, 10 ]
```

```
gap> Neighbors(cgr,1,[1,2]);
[ 1, 2, 3, 4, 5, 6, 7 ]
```

1.4.8 AdjacencyMatrix (first variant)

▷ `AdjacencyMatrix(cgr)` (method)
▷ `AdjacencyMatrix(cgr, colors)` (method)

Returns the adjacency matrix of *cgr*. If A is the adjacency matrix of *cgr*, then $A(i,j)$ is equal to the color (not to the color name!) of the arc (i,j).

Example

```
gap> m:=[["black","red"  ,"blue" ,"blue" ,"blue" ],
>        ["blue" ,"black","red"  ,"blue" ,"blue" ],
>        ["blue" ,"blue", "black","red"  ,"blue" ],
>        ["blue" ,"blue", "blue" ,"black","red"  ],
>        ["red"  ,"blue", "blue" ,"blue" ,"black"]];;
gap> cgr:=ColorGraphByMatrix(m);
<color graph of order 5 and rank 3>
gap> Display(AdjacencyMatrix(cgr));
[ [  1,  3,  2,  2,  2 ],
  [  2,  1,  3,  2,  2 ],
  [  2,  2,  1,  3,  2 ],
  [  2,  2,  2,  1,  3 ],
  [  3,  2,  2,  2,  1 ] ]
gap> ColorNames(cgr)
[ "black", "blue", "red" ]
```

In the second form `AdjacencyMatrix(cgr,colors)` returns a 0/1-matrix $A(i,j)$, that has entry 1 at (i,j) iff the entry of `AdjacencyMatrix(cgr)` at (i,j) is an element of the list *colors*.

Example

```
gap> Display(AdjacencyMatrix(cgr, [1,3]));
[ [  1,  1,  0,  0,  0 ],
  [  0,  1,  1,  0,  0 ],
  [  0,  0,  1,  1,  0 ],
  [  0,  0,  0,  1,  1 ],
  [  1,  0,  0,  0,  1 ] ]
```

1.4.9 RowOfColorGraph

▷ `RowOfColorGraph(cgr, i)` (operation)

Returns the i-th row of the adjacency matrix of *cgr* (`AdjacencyMatrix` (1.4.8)).

1.4.10 ColumnOfColorGraph

▷ `ColumnOfColorGraph(cgr, j)` (operation)

Returns the j-th column of the adjacency matrix of *cgr* (`AdjacencyMatrix` (1.4.8)).

1.4.11 Fibres

▷ Fibres(*cgr*) (operation)

The *Fibres* of a color graph are the maximal sets of vertices whose corresponding loops all have the same color.

Example
```
gap> cgr:=ColorGraph(SymmetricGroup(4), Combinations([1..4]), OnSets,
> true, functions(m1,m2) return Length(Intersection(m1,m2));end);
<color graph of order 16 and rank 5>
gap> Fibres(cgr);
[ [ 1 ], [ 2, 10, 14, 16 ], [ 3, 7, 9, 11, 13, 15 ], [ 4, 6, 8, 12 ], [ 5 ] ]
gap> VertexNamesOfCocoObject(cgr);
[ [  ], [ 1 ], [ 1, 2 ], [ 1, 2, 3 ], [ 1, 2, 3, 4 ], [ 1, 2, 4 ],
[ 1, 3 ], [ 1, 3, 4 ], [ 1, 4 ], [ 2 ], [ 2, 3 ], [ 2, 3, 4 ],
[ 2, 4 ], [ 3 ], [ 3, 4 ], [ 4 ] ]
```

1.4.12 NumberOfFibres

▷ NumberOfFibres(*cgr*) (attribute)

Returns the number of different colors of loops of *cgr* (cf. Fibres (1.4.11)).

1.4.13 LocalIntersectionArray

▷ LocalIntersectionArray(*cgr*, *v*, *w*) (method)
▷ LocalIntersectionArray(*cgr*, *arc*) (method)

The input to this operation is a color graph *cgr* and an arc. In the first version this arc is given as two parameters *v*, and *w*. In the second form the arc is given as ordered pair *arc*. We will assume in the following that arc=[v,w]. The local intersection array of the arc (v,w) is the square matrix A of order RankOfColorGraph(cgr) where $A(i,j)$ is equal to the number of vertices u of *cgr* such that the arc (v,u) has color i and the arc (u,w) has color j.

Example
```
gap> cgr:=JohnsonScheme(5,2);
<color graph of order 10 and rank 3>
gap> ColorRepresentative(cgr,1);
[ 1, 1 ]
gap> ColorRepresentative(cgr,2);
[ 1, 2 ]
gap> ColorRepresentative(cgr,3);
[ 1, 8 ]
gap> Display(LocalIntersectionArray(cgr,1,1));
[ [  1,  0,  0 ],
  [  0,  6,  0 ],
  [  0,  0,  3 ] ]
gap> Display(LocalIntersectionArray(cgr,1,2));
[ [  0,  1,  0 ],
  [  1,  3,  2 ],
  [  0,  2,  1 ] ]
```

```
gap> Display(LocalIntersectionArray(cgr,1,8));
[ [  0,  0,  1 ],
  [  0,  4,  2 ],
  [  1,  2,  0 ] ]
```

1.4.14 ColorMates

▷ ColorMates(*cgr*) (attribute)

In a WL-stable color graph for every color i there exists a color i' such that whenever an arc (u,v) has color i, then the opposite arc (v,u) has color i'. The mapping from i to i' is a permutation of the colors. The function ColorMates returns this permutation.

Example
```
gap> cgr:=ColorGraph(Group((1,2,3,4,5)));;
gap> Display(AdjacencyMatrix(cgr));
[ [  1,  2,  3,  4,  5 ],
  [  5,  1,  2,  3,  4 ],
  [  4,  5,  1,  2,  3 ],
  [  3,  4,  5,  1,  2 ],
  [  2,  3,  4,  5,  1 ] ]
gap> ColorMates(cgr);
(2,5)(3,4)
```

1.4.15 OutValencies (for WL-stable color graphs)

▷ OutValencies(*cgr*) (method)

Let i and a color of *cgr*. Then there is a number $d(i)$ such that for every vertex v of *cgr* there is either no arc, or there are exactly $d(i)$ arcs leaving v. The number $d(i)$ is called the *subdegree* of the color i.

The function OutValencies returns a the list [d(1),d(2),...,d(RankOfColorGraph(cgr))]

1.4.16 ReflexiveColors (for WL-stable color graphs)

▷ ReflexiveColors(*cgr*) (method)

This function returns the list of all reflexive colors of the WL-stable color graph *cgr*.

1.5 Creating new (color) graphs from given color graphs

1.5.1 ColorGraphByFusion

▷ ColorGraphByFusion(*cgr, fusion*) (operation)

The function takes as arguments a color graph *cgr* and a fusion. The fusion can be either a list of sets of colors, or it belongs to the category `IsFusionOfTensor` and more concretely to the family `FusionFamily(StructureConstantsOfColorGraph(cgr))`. In the latter case, *cgr* has to be WL-stable.

The fusion-color graph has the same order like *cgr*. The color of an arc (i, j) in the fusion color graph is the list of all classes of *fusion* to which `ArcColorOfColorGraph(cgr,i,j)` belongs. If *fusion* is a partition, then the effect is that all colors in one class are fused into the same new color. If *fusion* is not a partition, then the resulting color graph will be color-isomorphic to the fusion color graph of *cgr* with respect to the coarsest partition that allows to obtain every element of *fusion* as a union of classes.

Example

```
gap> cgr:=ColorGraph(Group((1,2,3,4,5)));
<color graph of order 5 and rank 5>
gap> cgr2:=ColorGraphByFusion(cgr,[[1],[2,3],[4],[5]]);
<color graph of order 5 and rank 4>
gap> Display(AdjacencyMatrix(cgr));
[ [  1,  2,  3,  4,  5 ],
  [  5,  1,  2,  3,  4 ],
  [  4,  5,  1,  2,  3 ],
  [  3,  4,  5,  1,  2 ],
  [  2,  3,  4,  5,  1 ] ]
gap> Display(AdjacencyMatrix(cgr2));
[ [  1,  2,  2,  3,  4 ],
  [  4,  1,  2,  2,  3 ],
  [  3,  4,  1,  2,  2 ],
  [  2,  3,  4,  1,  2 ],
  [  2,  2,  3,  4,  1 ] ]
gap> ColorNames(cgr2);
[ [ [ 1 ] ], [ [ 2, 3 ] ], [ [ 4 ] ], [ [ 5 ] ] ]
```

1.5.2 QuotientColorGraph

▷ `QuotientColorGraph(`*cgr, part*`)` (operation)

part is a partition of the vertex set of the color graph *cgr* (it has to be a set of sets of vertices). The quotient graph of *cgr* with respect to *part* has as vertex set the classes of *part*. the color of the arc $([u],[v])$ the quotient graph is the set of all colors i of *cgr* such that there are vertices $u' \in [u]$ and $v' \in [v]$ such that the arc (u',v') has color i.

The above described color graph is also well-defined, if *part* is not a partition but any set of sets of vertices of *cgr*. In fact, `QuotientColorGraph` does not check, whether *part* is indeed a partition.

Example

```
gap> s5:=SymmetricGroup(5);;
gap> cgr:=ColorGraph(s5, Arrangements([1..5],2), OnPairs,true);
<color graph of order 20 and rank 7>
gap> part:=Set(Orbit(s5, [[1,2],[2,1]], OnSetsTuples));;
gap> part:=Set(part, x->Set(x, y->Position(VertexNamesOfCocoObject(cgr),y)));
[ [ 1, 5 ], [ 2, 9 ], [ 3, 13 ], [ 4, 17 ], [ 6, 10 ], [ 7, 14 ],
[ 8, 18 ], [ 11, 15 ], [ 12, 19 ], [ 16, 20 ] ]
gap> cgr2:=QuotientColorGraph(cgr,part);
```

```
<color graph of order 10 and rank 3>
gap> ColorNames(cgr2);
[ [ 1, 3 ], [ 2, 4, 5, 6 ], [ 7 ] ]
gap> VertexNamesOfCocoObject(cgr2);
[ [ 1, 5 ], [ 2, 9 ], [ 3, 13 ], [ 4, 17 ], [ 6, 10 ], [ 7, 14 ],
[ 8, 18 ], [ 11, 15 ], [ 12, 19 ], [ 16, 20 ] ]
```

1.5.3 InducedSubColorGraph

▷ `InducedSubColorGraph(`*cgr, set*`)` (operation)

This function returns a color graph that is isomorphic to the sub color graph induced by *set*. The function that maps i to `set[i]` is an embedding of the induced subgraph into *cgr*.

Example

```
gap> cgr:=ColorGraph(SymmetricGroup(5), Combinations([1..5]), OnSets, true);
<color graph of order 32 and rank 56>
gap> vn:=VertexNamesOfCocoObject(cgr);;
gap> fibre:=Filtered([1..Length(vn)], i->Length(vn[i])=2);
[ 3, 11, 15, 17, 19, 23, 25, 27, 29, 31 ]
gap> cgr2:=InducedSubColorGraph(cgr,fibre);
<color graph of order 10 and rank 3>
gap> VertexNamesOfCocoObject(cgr2);
[ 3, 11, 15, 17, 19, 23, 25, 27, 29, 31 ]
```

1.5.4 DirectProductColorGraphs

▷ `DirectProductColorGraphs(`*cgr1, cgr2*`)` (operation)

Suppose, *cgr1* is the color graph (V_1,C_1,f_1), and *cgr2* is the color graph (V_2,C_2,f_2). Then the direct product of *cgr1* with *cgr2* has vertex set $V_1 \times V_2$, and color set $C_1 \times C_2$. The coloring function is $f_1 \times f_2$. Here $f_1 \times f_2$ acts coordinate wise.

The operation `DirectProductColorGraphs` returns the direct product of *cgr1* with *cgr2*.

1.5.5 WreathProductColorGraphs

▷ `WreathProductColorGraphs(`*cgr1, cgr2*`)` (operation)

Suppose, *cgr1* is the color graph (V_1,C_1,f_1), and *cgr2* is the color graph (V_2,C_2,f_2). Suppose, D_1 is the set of all those colors of *cgr1* whose color class contains reflexive tuples. Then the wreath product of *cgr1* with *cgr2* has vertex set $V_1 \times V_2$. The set of colors is the union of $C_1 \times \{*\}$ with $D_1 \times C_2$. The coloring function maps pairs $((a_1,a_2),(a_1,b_2))$ to $(f_1(a_1,a_1),f_2(b_1,b_2))$, and other pairs $((a_1,a_2),(b_1,b_2))$ to $(f_1(a_1,a_2),*)$.

The operation `WreathProductColorGraphs` returns the wreath product of *cgr1* with *cgr2*.

1.5.6 ClosedSets (for homogeneous WL-stable color graphs)

▷ `ClosedSets(cgr)` (attribute)

A set `cset` of colors of *cgr* is closed if the collections of all arcs whose color is from `cset` forms an equivalence relation. This function returns a list of all closed sets of colors of *cgr*.

1.5.7 PartitionClosedSet (for homogeneous WL-stable color graphs)

▷ `PartitionClosedSet(cgr, cset)` (operation)

A set *cset* of colors of *cgr* is closed if the collections of all arcs whose color is from *cset* forms an equivalence relation. This function returns the vertex-partition corresponding to this equivalence relation. It is not tested, whether *cset* is indeed closed. It is required that *cgr* is a homogeneous WL-stable color graph.

Example

```
gap> s5:=SymmetricGroup(5);;
gap> d6:=Subgroup(s5, [(1,2),(1,2,3)(4,5)]);;
gap> cgr:=ColorGraph(s5,s5,OnRight,true, function(a,b) return a*b;end);
<color graph of order 120 and rank 120>
gap> cset:=Set(d6, x->Position(ColorNames(cgr),x));
[ 1, 8, 13, 24, 29, 31, 61, 68, 73, 84, 89, 91 ]
gap> IsWLStableColorGraph(cgr);
true
gap> IsHomogeneous(cgr);
true
gap> part:=PartitionClosedSet(cgr,cset);;
gap> cgr2:=QuotientColorGraph(cgr,part);
<color graph of order 10 and rank 3>
```

1.5.8 BaseGraphOfColorGraph (first variant)

▷ `BaseGraphOfColorGraph(cgr, color)` (method)
▷ `BaseGraphOfColorGraph(cgr, cset)` (method)

This function extracts graphs from a color graph. In the first variant, the second argument is one color. In this case the digraph with vertex set `[1..OrderOfColorGraph(cgr)]` and with all arcs of color *color* from *cgr*.

In the second case the arc-set of the result consists of all arcs with color from *cset* of *cgr*.

This function is available only if Grape is loaded.

Example

```
gap> cgr:=JohnsonScheme(5,2);
<color graph of order 10 and rank 3>
gap> OutValencies(cgr);
[ 1, 6, 3 ]
gap> gamma:=BaseGraphOfColorGraph(cgr,3);;
gap> IsDistanceRegular(gamma);
true
gap> GlobalParameters(gamma);
```

```
[ [ 0, 0, 3 ], [ 1, 0, 2 ], [ 1, 2, 0 ] ]
```

1.6 Testing properties of color graphs

1.6.1 IsUndirectedColorGraph

▷ `IsUndirectedColorGraph(cgr)` (property)

A color graph is called *undirected* if for all vertices u and v the arc (u,v) has the same color as the arc (v,u). The function tests this property for `cgr`.

Example

```
gap> cgr:=ColorGraph(Group((1,2,3,4,5)));
<color graph of order 5 and rank 5>
gap> IsUndirectedColorGraph(cgr);
false
gap> ArcColorOfColorGraph(cgr,[1,2]);
2
gap> ArcColorOfColorGraph(cgr,[2,1]);
5
gap> cgr2:=ColorGraphByFusion(cgr, [[1],[2,5],[3,4]]);
<color graph of order 5 and rank 3>
gap> IsUndirectedColorGraph(cgr2);
true
```

1.6.2 IsHomogeneous

▷ `IsHomogeneous(cgr)` (property)

A color graph is homogeneous if it has just one fibre. For a WL-stable color graph this means that it has just one reflexive color. For a WL-stable color graph this means that it correspond to an association scheme.

Example

```
gap> e8:=ElementaryAbelianGroup(8);
<pc group of size 8 with 3 generators>
gap> e8:=Action(e8,AsList(e8), OnRight);
Group([ (1,2)(3,5)(4,6)(7,8), (1,3)(2,5)(4,7)(6,8), (1,4)(2,6)(3,7)(5,8) ])
gap> cgr:=ColorGraph(e8,Combinations([1..DegreeAction(g)],2), OnSets);
<color graph of order 28 and rank 112>
gap> IsHomogeneous(cgr);
false
```

1.6.3 IsWLStableColorGraph

▷ `IsWLStableColorGraph(cgr)` (property)

This function returns true if *cgr* is stable under the Weisfeiler-Leman algorithm, that is, whether it is the color graph of a coherent configuration.

Example
```
gap> cgr:=ColorGraph(Center(GL(2,7)), GF(7)^2, OnRight, true,
> function(a,b) return NormedRowVector(a-b);end);
<color graph of order 49 and rank 9>
gap> IsWLStableColorGraph(cgr);
true
```

1.6.4 IsSchurian

▷ `IsSchurian(cgr)` (property)

A color graph is called *Schurian* if it is color isomorphic to the color graph of orbitals of its automorphism group.

Example
```
gap> lcgr:=AllAssociationSchemes(15);;
gap> lcgr:=Filtered(lcgr, x->not IsSchurian(x));
[ AS(15,5) ]
```

1.6.5 IsPrimitive (for WL-stable color graphs)

▷ `IsPrimitive(cgr)` (property)

A WL-stable color graph is primitive if all its loopless base graphs are strongly connected (cf. `BaseGraphOfColorGraph` (1.5.8)). This function tests, whether *cgr* is primitive or not.

Example
```
gap> cgr:=ColorGraph(Group((1,2,3,4)));
<color graph of order 4 and rank 4>
gap> IsPrimitive(cgr);
false
gap> ReflexiveColors(cgr);
[ 1 ]
gap> IsConnectedGraph(BaseGraphOfColorGraph(cgr,2));
true
gap> IsConnectedGraph(BaseGraphOfColorGraph(cgr,3));
false
gap> IsConnectedGraph(BaseGraphOfColorGraph(cgr,4));
true
```

1.7 Symmetries of color graphs

1.7.1 KnownGroupOfAutomorphisms (for color graphs)

▷ `KnownGroupOfAutomorphisms(cgr)` (operation)

This function returns the group of all automorphisms of *cgr* that COCO2P knows at the given moment.

1.7.2 AutGroupOfCocoObject (for color graphs)

▷ `AutGroupOfCocoObject(cgr)` (attribute)
▷ `AutomorphismGroup(cgr)` (method)

Returns the group of all permutations of the vertices of *cgr* that preserve the color of all arcs.

1.7.3 IsAutomorphismOfObject (for color graphs)

▷ `IsAutomorphismOfObject(cgr, perm)` (operation)
▷ `IsAutomorphismOfColorGraph(cgr, perm)` (operation)

Returns `true`, if *perm* is an automorphism of *cgr*. In that case COCO2P adds *perm* to the known automorphisms of *cgr*.

1.7.4 IsomorphismCocoObjects (for color graphs)

▷ `IsomorphismCocoObjects(cgr1, cgr2)` (operation)
▷ `IsomorphismColorGraphs(cgr1, cgr2)` (operation)

An isomorphism from *cgr1* to *cgr2* is a bijection between the vertex sets that preserves the color of arcs (including the names of colors).

This operation returns an isomorphism from *cgr1* to *cgr2* if it exists, and `fail` if it does not exists.

1.7.5 IsIsomorphicCocoObject (for color graphs)

▷ `IsIsomorphicCocoObject(cgr1, cgr2)` (operation)
▷ `IsIsomorphicColorGraph(cgr1, cgr2)` (operation)

Returns `true` if *cgr1* and *cgr2* are isomorphic, and `false` otherwise (cf. `IsomorphismCocoObjects` (1.7.4))

1.7.6 IsIsomorphismOfObjects (for color graphs)

▷ `IsIsomorphismOfObjects(cgr1, cgr2, g)` (operation)
▷ `IsIsomorphismOfColorGraphs(cgr1, cgr2, g)` (operation)

Returns `true` if *g* is an isomorphism fro *cgr1* to *cgr2*, and `false` otherwise (cf. `IsomorphismCocoObjects` (1.7.4)).

1.7.7 KnownGroupOfColorAutomorphisms

▷ `KnownGroupOfColorAutomorphisms(cgr)` (operation)

This function returns the group of all color automorphisms of *cgr* that COCO2P knows at the given moment.

1.7.8 LiftToColorAutomorphism

▷ `LiftToColorAutomorphism(`*cgr, perm*`)` (operation)

cgr is a color graph and *perm* is a permutation of its colors. The function constructs a color automorphism of *cgr* that acts like *perm* on the colors. If such a color automorphism does not exist, then `fail` is returned.

If *perm* is liftable, then the result of the lifting is added to the known group of color automorphisms of *cgr*.

1.7.9 LiftToColorIsomorphism

▷ `LiftToColorIsomorphism(`*cgr1, cgr2, ciso*`)` (operation)

cgr1 and *cgr2* are color graphs of the same rank, and *ciso* is a bijection from the colors of *cgr1* to the colors of *cgr2*. The function constructs a color isomorphism from *cgr1* to *cgr2* that acts like *ciso* on the colors. If such a color isomorphism does not exist, then `fail` is returned.

1.7.10 ColorIsomorphismColorGraphs

▷ `ColorIsomorphismColorGraphs(`*cgr1, cgr2*`)` (operation)

This operation returns a color isomorphism from *cgr1* to *cgr2*, and `fail` otherwise.

At the moment, this operation is implemented only from WL-stable color graphs.

1.7.11 IsColorIsomorphicColorGraph

▷ `IsColorIsomorphicColorGraph(`*cgr1, cgr2*`)` (operation)

This operation returns `true` if *cgr1* and *cgr2* are color isomorphic, and `false` otherwise.

At the moment, this operation is implemented only from WL-stable color graphs.

1.7.12 ColorAutomorphismGroup

▷ `ColorAutomorphismGroup(`*cgr*`)` (attribute)

This function computes and returns the color automorphism group of *cgr*. This group consists of all permutations of the vertices of the color graph, that map arcs of the same color to arcs of the same color. In particular, it may act non-trivially on the colors of *cgr*.

If *cgr* is a Schurian WL-stable color graph, then its color automorphism group is equal to the normalizer of its automorphism group in the full symmetric group of the vertices of *cgr*. In some (rare) cases, this way to compute normalizers can be quicker than the built-in gap-functions.

At the moment, this function is implemented only for WL-stable color graphs.

1.7.13 ColorAutomorphismGroupOnColors

▷ `ColorAutomorphismGroupOnColors(`*cgr*`)` (attribute)

The color automorphism of *cgr* acts on the colors of *cgr* with the automorphism group of *cgr* as kernel. This function computes and returns this action.

At the moment, this function is implemented only for WL-stable color graphs.

1.7.14 KnownGroupOfAlgebraicAutomorphisms

▷ `KnownGroupOfAlgebraicAutomorphisms(`*cgr*`)` (operation)

This function returns the group of all algebraic automorphisms of *cgr* that COCO2P knows at the given moment.

1.7.15 AlgebraicAutomorphismGroup

▷ `AlgebraicAutomorphismGroup(`*cgr*`)` (attribute)

The algebraic automorphism group of a WL-stable color graph is nothing but the automorphism group of its tensor of structure constants. The color automorphism group in its action on colors embeds naturally into the algebraic automorphism group.

Chapter 2

Structure Constants Tensors

2.1 Introduction

COCO2P introduces its own data-type for structure constants tensors of coherent algebras. The methods provided by COCO2P are tailored for this use. The emphasis lies on symmetries, quotients (by closed sets) and mergings (fusions).

2.2 Functions for the construction of tensors

2.2.1 StructureConstantsOfColorGraph

▷ `StructureConstantsOfColorGraph(`*cgr*`)` (attribute)

This function expects a WL-stable color graph *cgr*, and computes its tensor of structure constants. The result is the structure constants tensor T of *cgr*. This object encodes a third-order tensor. For every color k of *cgr*, the matrix $T(i,j,k)$ is equal to the `LocalIntersectionArray` (1.4.13) of any arc of color k in *cgr*.

2.2.2 DenseTensorFromEntries

▷ `DenseTensorFromEntries(`*entries*`)` (function)

The argument *entries* is a list of lists of lists of integers. There has to be a number n such that `Length(entries)=n`, for all $1 \leq i, j \leq n$ `Length(entries[i])=n`, and `Length(entries[i][j]=n`. Otherwise there are no restrictions.

The function returns the tensor-object for *entries*.

Note that this function does not check, whether the entries are integers or even numbers. One can also view the datatype of tensors as a type that encodes complete colored hyper-graphs with hyper-arcs of length 3. Even though there is not much infrastructure implemented in COCO2P for such objects, at least it is possible to check isomorphism and to compute automorphism groups.

2.3 Functions for the inspection of tensors

2.3.1 OrderOfTensor

▷ `OrderOfTensor(tensor)` (attribute)
▷ `OrderOfCocoObject(tensor)` (attribute)
▷ `Order(tensor)` (attribute)

Returns the order of the tensor. If it is equal to n then this means that *tensor* is an $n \times n \times n$-array.

2.3.2 VertexNamesOfCocoObject (for tensors)

▷ `VertexNamesOfCocoObject(tensor)` (operation)
▷ `VertexNamesOfTensor(tensor)` (operation)

Returns the list of names of the vertices of *tensor*.

2.3.3 EntryOfTensor

▷ `EntryOfTensor(tensor, i, j, k)` (operation)

Returns the entry at index (i, j, k) of *tensor*.

2.3.4 ReflexiveColors (for structure constants tensors)

▷ `ReflexiveColors(tensor)` (attribute)

If *tensor* is the structure constants tensor of the WL-stable color graph `cgr`, then `ReflexiveColors(tensor)` return the list of all reflexive colors of `cgr`.

Example

```
gap> e8:=Action(e8,AsList(e8), OnRight);
Group([ (1,2)(3,5)(4,6)(7,8), (1,3)(2,5)(4,7)(6,8), (1,4)(2,6)(3,7)(5,8) ])
gap> cgr:=ColorGraph(e8,Combinations([1..DegreeAction(g)],2), OnSets);
<color graph of order 28 and rank 112>
gap> T:=StructureConstantsOfColorGraph(cgr);
<Tensor of order 112>
gap> ReflexiveColors(T);
[ 1, 18, 35, 52, 69, 86, 103 ]
```

2.3.5 NumberOfFibres (for structure constants tensors)

▷ `NumberOfFibres(tensor)` (attribute)

Returns the number of reflexive colors of *tensor*.

2.3.6 FibreLengths (for structure constants tensors)

▷ `FibreLengths(tensor)` (attribute)

If *tensor* is the structure constants tensor of the WL-stable color graph `cgr`, then `FibreLengths(tensor)` returns the list of lengths of all fibres of `cgr`. The order corresponds to the result of `ReflexiveColors(T)`.

Example
```
gap> a5:=AlternatingGroup(5);
Alt( [ 1 .. 5 ] )
gap> g:=Action(a5, Combinations([1..5],2), OnSets);
Group([ (1,5,8,10,4)(2,6,9,3,7), (2,3,4)(5,6,7)(8,10,9) ])
gap> g:=Stabilizer(g,1);
Group([ (2,3,4)(5,6,7)(8,10,9), (2,6)(3,5)(4,7)(9,10) ])
gap> cgr:=ColorGraph(g);
<color graph of order 10 and rank 19>
gap> T:=StructureConstantsOfColorGraph(cgr);
<Tensor of order 19>
gap> ReflexiveColors(T);
[ 1, 5, 18 ]
gap> FibreLengths(T);
[ 1, 6, 3 ]
gap> Fibres(cgr);
[ [ 1 ], [ 2, 3, 4, 5, 6, 7 ], [ 8, 9, 10 ] ]
```

2.3.7 OutValencies (for structure constants tensors)

▷ `OutValencies(tensor)` (attribute)

If *tensor* is the structure constants tensor of the WL-stable color graph `cgr`, then `OutValencies(tensor)` returns the `OutValencies` (1.4.15) of *cgr*.

2.3.8 Mates (for structure constants tensors)

▷ `Mates(tensor)` (attribute)

If *tensor* is the structure constants tensor of the WL-stable color graph `cgr`, then `OutValencies(tensor)` returns the permutation `ColorMates(cgr)` (`ColorMates` (1.4.14)).

2.3.9 StartBlock (for structure constants tensors)

▷ `StartBlock(tensor, i)` (operation)

If *tensor* is the structure constants tensor of the WL-stable color graph `cgr`, then in particular, the vertices of *tensor* are the colors of `cgr`. All arcs of color *i* have their starting vertex in the same fibre of `cgr`. Moreover, the loops over the vertices of one fibre all have the same color.

This function returns the index j into `ReflexiveColors(T)` (cf. `ReflexiveColors` (2.3.4)) such that at the start of every arc of color *i* there is a loop to color `ReflexiveColors(T)[j]`.

2.3.10 FinishBlock (for structure constants tensors)

▷ `FinishBlock(`*`tensor, i`*`)` (operation)

If *`tensor`* is the structure constants tensor of the WL-stable color graph `cgr`, then in particular, the vertices of *`tensor`* are the colors of `cgr`. All arcs of color *`i`* have their finishing vertex in the same fibre of `cgr`. Moreover, the loops over the vertices of one fibre all have the same color.

This function returns the index j into `ReflexiveColors(T)` (cf. `ReflexiveColors` (2.3.4)) such that at the end of every arc of color *`i`* there is a loop to color `ReflexiveColors(T)[j]`.

2.3.11 ClosedSets

▷ `ClosedSets(`*`tensor`*`)` (operation)

A set M of vertices of *`tensor`* is called *closed* if whenever i, j are in M, then also all such k are in M for which `EntryOfTensor(tensor,i,j,k)` is non-zero.

This function returns all closed sets of *`tensor`*.

2.3.12 ComplexProduct (for structure constants tensors)

▷ `ComplexProduct(`*`tensor, set1, set2`*`)` (operation)

Suppose that *`tensor`* is the structure constants tensor of the WL-stale color graph `cgr`. the colors of `cgr` canonically correspond to the standard-basis elements of the coherent algebra W that is associated with *`cgr`*. The elements of W can naturally be encoded as vectors of length `Rank(cgr)`. The arguments *`set1`* and *`set2`* are sets of colors of `cgr` (i.e. vertices of *`tensor`*). Their characteristic vectors, can hence be understood as elements of W.

The operation `ComplexProduct` returns the coefficient-vector of the product of the characteristic vector of *`set1`* with the characteristic vector of *`set2`* in W.

2.3.13 ClosureSet

▷ `ClosureSet(`*`tensor, set`*`)` (function)

`set` is a set of vertices of *`tensor`*. The function returns the smallest closed set of *`tensor`* that contains *`set`* (cf. `ClosedSets` (2.3.11))

2.4 Testing properties of tensors

2.4.1 IsTensorOfCC

▷ `IsTensorOfCC(`*`tensor`*`)` (property)

If *`tensor`* has this property, then this means that COCO2P knows, that it is the structure constants tensor of a WL-stable color graph. There is no method installed for this property, as it is in general hard to prove that a given tensor belongs to a WL-stable color graph. The property is set by the constructor that created *`tensor`*.

2.4.2 IsCommutativeTensor

▷ `IsCommutativeTensor(`*tensor*`)` (property)

tensor has this property, if for all i,j,k holds `EntryOfTensor(tensor,i,j,k)= EntryOfTensor(tensor,j,i,k)`.

2.4.3 IsHomogeneous (for structure constants tensors)

▷ `IsHomogeneous(`*tensor*`)` (property)

Returns whether *tensor* has just one reflexive color.

2.4.4 IsPrimitive (for structure constants tensors)

▷ `IsPrimitive(`*tensor*`)` (property)

A structure constants tensor is *primitive* if it is homogeneous and if it has only the trivial closed sets (i.e. the singleton of the unique reflexive color and the set of all colors).

If *tensor* is the structure constants tensor of the color graph `cgr`, then *tensor* is primitive if and only if `cgr` is primitive (cf. `IsPrimitive` (1.6.5)).

2.5 Symmetries of tensors

2.5.1 KnownGroupOfAutomorphisms (for tensors)

▷ `KnownGroupOfAutomorphisms(`*tensor*`)` (operation)

This function returns the group of all automorphisms of *tensor* that COCO2P knows at the given moment.

2.5.2 AutGroupOfCocoObject (for tensors)

▷ `AutGroupOfCocoObject(`*tensor*`)` (attribute)
▷ `AutomorphismGroup(`*tensor*`)` (method)

Returns the group of all automorphisms of *tensor*.

2.5.3 IsAutomorphismOfObject (for tensors)

▷ `IsAutomorphismOfObject(`*tensor, perm*`)` (operation)
▷ `IsAutomorphismOfTensor(`*tensor, perm*`)` (operation)

Returns `true`, if *perm* is an automorphism of *tensor*. In that case COCO2P adds *perm* to the known automorphisms of *tensor*.

2.5.4 IsomorphismCocoObjects (for tensors)

▷ `IsomorphismCocoObjects(tensor1, tensor2)` (operation)
▷ `IsomorphismTensors(tensor1, tensor2)` (operation)

This operation returns an isomorphism from *tensor1* to *tensor2* if it exists, and `fail` if it does not exists.

2.5.5 IsIsomorphicCocoObject (for tensors)

▷ `IsIsomorphicCocoObject(tensor1, tensor2)` (operation)
▷ `IsIsomorphicTensor(tensor1, tensor2)` (operation)

Returns `true` if *tensor1* and *tensor2* are isomorphic, and `false` otherwise.

2.5.6 IsIsomorphismOfObjects (for tensors)

▷ `IsIsomorphismOfObjects(tensor1, tensor2, g)` (operation)
▷ `IsIsomorphismOfTensors(tensor1, tensor2, g)` (operation)

Returns `true` if *g* is an isomorphism fro *tensor1* to *tensor2*, and `false` otherwise.

2.6 Character tables of structure constants tensors

The structure constants tensor of a WL-stable color graph encodes the structure of the associated coherent algebra. If this algebra is commutative, then COCO2P is able to compute its character table provided, the irrationalities occuring are representable in GAP. The algorithm that computes the character tables involves Gröbner-bases. The computation of the Gröbner bases defines the overall performance of the algorithm for the computation of character tables.

2.6.1 CharacterTableOfTensor (for commutative structure constants tensors)

▷ `CharacterTableOfTensor(tensor)` (attribute)

This function returns a record with two components: `characters` and `multiplicities`. If `ct` is the character table of *tensor*, then `ct.characters[i][j]` is the value of the i-th irreducible character of the standard-basis element corresponding to color j of *tensor*. Moreover, `ct.multiplicities[i]` is the multiplicity of the i-th irreducible character.

Example

```
gap> cgr:=JohnsonScheme(6,3);
<color graph of order 20 and rank 4>
gap> T:=StructureConstantsOfColorGraph(cgr);
<Tensor of order 4>
gap> IsCommutativeTensor(T);
true
gap> CharacterTable(T);
rec( characters := [ [ 1, 9, 9, 1 ], [ 1, -1, -1, 1 ], [ 1, -3, 3, -1 ],
      [ 1, 3, -3, -1 ] ], multiplicities := [ 1, 9, 5, 5 ] )
```

Chapter 3

WL-Stable Fusions Color Graphs

3.1 Introduction

One of the fundamental methods how to derive new color graphs from a color graph Γ, is to *fuse* (i.e identify) colors. Color graphs that are derived from Γ in this way are called *fusion color graphs*. Every fusion color graph Δ of Γ defines a partition on the colors of Γ. This partition is called the *fusion* associated with the fusion color graph Δ of Γ. If Δ is WL-stable, then its fusion is called a *stable fusion*.

One of the fundamental algorithmical problems in algebraic combinatorics is to enumerate all WL-stable fusion color graphs of a given color graph. At the moment COCO2P can solve a part of this problem – namely starting from any WL-stable color graph Γ it can enumerate (orbits of) stable fusions that lead to homogeneous WL-stable fusion color graphs. Such fusions we will call *homogeneous*.

Computing stable fusions, in COCO2P is a two-stages process:

1. Computation of good sets of colors,

2. Fitting together good sets to stable fusions.

Good sets are the building blocks of stable fusions. A set of colors of a WL-stable color graph is called a *good set* if there exists a stable fusion of the cgr in which the set appears as a class. It is called a *homogeneous good set* if it is part of a homogeneous stable fusion. Note that the property to be a (homogeneous) good set does only depend on the structure constants of the color graph.

3.2 Good sets

3.2.1 BuildGoodSet

▷ `BuildGoodSet(`*`tensor, set[, part]`*`)` (function)

`tensor` is the structure constants tensor of a WL-stable color graph `cgr`. *`set`* is a set of colors of `cgr` (i.e. of vertices of *`tensor`*). *`part`* is supposed to be the coarsest stable partition of the colors of `cgr` that contains *`set`* as a class (the stability is not checked by the function). The function returns the corresponding good-set object.

If *`part`* is not given, then it is computed. If this computation fails (because *`set`* is not a good set), then `fail` is returned.

3.2.2 AsSet (for good sets)

▷ `AsSet(gs)` (operation)

Converts the good set object *gs* into a usual set.

3.2.3 TensorOfGoodSet

▷ `TensorOfGoodSet(gs)` (operation)

Returns the structure constants tensor over which the good set *gs* is “good”.

3.2.4 PartitionOfGoodSet

▷ `PartitionOfGoodSet(gs)` (operation)

This function returns the coarsest stable fusion (as a partition, i.e. a set of sets of colors), that contains *gs* as a class.

3.3 Orbits of good sets

COCO2P implements a datatype for orbits of combinatorial objects. This section describes the functions that deal with orbits of good sets. For every orbit of good sets, only the lexicographically smallest representative and its set-wise stabilizer is saved. This allows to deal with good sets of color graphs of comparatively high rank, provided they have many algebraic automorphisms.

3.3.1 HomogeneousGoodSetOrbits (for structure constants tensors)

▷ `HomogeneousGoodSetOrbits(tensor)` (attribute)
▷ `HomogeneousGoodSetOrbits(group, tensor[, mode])` (method)

group is supposed to consist only of automorphisms of *tensor*. COCO2P learns new automorphisms by checking this property. If group is not given, then the full automorphism group of *tensor* is taken for *group*.

This function returns all *group*-orbits of homogeneous good sets.

If *mode* is equal to `"s"`, then only orbits of symmetric good sets are returned. If *mode* is equal to `"a"`, then only orbits of asymmetric good sets are returned.

3.3.2 GoodSetOrbit

▷ `GoodSetOrbit(group, gs[, stab])` (operation)

gs is a good set. *group* has to be a subgroup of the automorphism group of `TensorOfGoodSet(gs)`. *stab* (if given) has to be the full set-wise stabilizer of *gs* in *group*.

The function constructs a COCO2P-orbit object of the setwise orbit of *gs* under *group*.

3.3.3 CanonicalRepresentativeOfCocoOrbit (for orbits of good sets)

▷ `CanonicalRepresentativeOfCocoOrbit(`*gsorb*`)` (operation)

This function returns the lexicographically smallest element of the orbit of good sets *gsorb*.

3.3.4 Representative (for orbits of good sets)

▷ `Representative(`*gsorb*`)` (operation)

This function returns any element of the orbit of good sets *gsorb*. At the moment it in fact returns the lexicographically smallest element.

3.3.5 UnderlyingGroupOfCocoOrbit (for orbits of good sets)

▷ `UnderlyingGroupOfCocoOrbit(`*gsorb*`)` (operation)

This function returns the group under which *gsorb* is an orbit.

3.3.6 StabilizerOfCanonicalRepresentative (for orbits of good sets)

▷ `StabilizerOfCanonicalRepresentative(`*gsorb*`)` (operation)

This function returns the setwise stabilizer of `CanonicalRepresentativeOfCocoOrbit(gsorb)` in `UnderlyingGroupOfCocoOrbit(gsorb)`.

3.3.7 Size (for orbits of good sets)

▷ `Size(`*gsorb*`)` (method)

returns the size of *gsorb*.

3.3.8 AsList (for orbits of good sets)

▷ `AsList(`*gsorb*`)` (method)

expands the COCO2P-orbit object *gsorb* into a list of good sets.

3.3.9 AsSet (for orbits of good sets)

▷ `AsSet(`*gsorb*`)` (method)

expands the COCO2P-orbit object *gsorb* into a set of good sets.

3.3.10 SubOrbitsOfCocoOrbit (for orbits of good sets)

▷ `SubOrbitsOfCocoOrbit(group, gsorb)` (operation)

group is a subgroup of the underlying group of the orbit of good sets *gsorb*. The given orbit splits into suborbits under this group. The function returns a list of these suborbits.

3.3.11 SubOrbitsWithInvariantPropertyOfCocoOrbit (for orbits of good sets)

▷ `SubOrbitsWithInvariantPropertyOfCocoOrbit(group, gsorb, prop)` (operation)

prop is a function that takes a single good set as argument and returns `true` or `false`. It has to be invariant under the set-wise action of *group*. Note that this property is not checked by the function.

This function does the same as

```
Filtered(SubOrbitsOfCocoOrbit(group,gsorb), x->prop(Representative(x)));
```

However, the former code is generally much less efficient than calling

```
SubOrbitsWithInvariantPropertyOfCocoOrbit(group,gsorb,prop);
```

3.4 Fusions

3.4.1 FusionFromPartition (for structure constant tensors)

▷ `FusionFromPartition(tensor, part)` (function)

if *tensor* is the structure constants tensor of the WL-stable color graph `cgr`, and if *part* is a partition of the colors of `cgr` (a set of sets of colors), then this function returns a fusion-object, or `fail` if *part* is not a fusion of `cgr`.

3.4.2 AsPartition

▷ `AsPartition(fusion)` (attribute)

Converts the fusion-object *fusion* into a set of sets of colors.

3.4.3 PartitionOfFusion

▷ `PartitionOfFusion(fusion)` (operation)

Converts the fusion object *fusion* into a list of sets. In contrast to te result of `AsPartition(fusion)`, the resulting list of classes is sorted in short-lex order. This means that first it is sorted by cardinality of classes, and then the classes of equal size are sorted lexicographically.

3.4.4 TensorOfFusion

▷ `TensorOfFusion(fusion)` (operation)

returns the structure constants tensor, over which the fusion `fusion` is a stable fusion.

3.4.5 BaseOfFusion

▷ `BaseOfFusion(fusion)` (attribute)

The base of a fusion is the smallest list of classes of `fusion` (in the short lex order) that generates `fusion` in the sense that there is no coarser stable fusion that contains the classes of the base.

This function returns the base of `fusion` if COCO2P knows it. At the moment there is no method for computing the base.

3.4.6 RankOfFusion

▷ `RankOfFusion(fusion)` (attribute)

returns the number of classes of `fusion`.

3.5 Orbits of fusions

COCO2P implements a datatype for orbits of combinatorial objects. This section describes the functions that deal with orbits of stable fusion. For every orbit of fusions, only the smallest representative in the short-lex order and its partition-wise stabilizer is saved. This allows to deal with fusions of color graphs of comparatively high rank.

3.5.1 HomogeneousFusionOrbits (for structure constants tensors)

▷ `HomogeneousFusionOrbits(tensor)` (attribute)
▷ `HomogeneousFusionOrbits(group, tensor)` (method)

`group` is supposed to consist only of automorphisms of `tensor`. COCO2P learns new automorphisms by checking this property. If group is not given, then the full automorphism group of `tensor` is taken for `group`.

This function returns all `group`-orbits of homogeneous stable fusions.

3.5.2 FusionOrbit

▷ `FusionOrbit(group, fusion[, stab])` (operation)

`fusion` is a fusion object. `group` has to be a subgroup of the automorphism group of `TensorOfFusion(fusion)`. `stab` (if given) has to be the full partition-wise stabilizer of `fusion` in `group`.

The function constructs a COCO2P-orbit object of the partition-wise orbit of `fusion` under `group`.

3.5.3 CanonicalRepresentativeOfCocoOrbit (for orbits of fusions)

▷ `CanonicalRepresentativeOfCocoOrbit(`*fusionorb*`)` (operation)

This function returns the smallest element (in the short-lex order) of the orbit of fusions *fusionorb*.

3.5.4 Representative (for orbits of fusions)

▷ `Representative(`*fusionorb*`)` (operation)

This function returns any element of the orbit of fusions sets *fusionorb*. At the moment it in fact returns the canonical representative.

3.5.5 UnderlyingGroupOfCocoOrbit (for orbits of fusions)

▷ `UnderlyingGroupOfCocoOrbit(`*fusionorb*`)` (operation)

This function returns the group under which *fusionorb* is an orbit.

3.5.6 StabilizerOfCanonicalRepresentative (for orbits of fusions)

▷ `StabilizerOfCanonicalRepresentative(`*fusion*`)` (operation)

This function returns the partition-wise stabilizer of `CanonicalRepresentativeOfCocoOrbit(fusionorb)` in `UnderlyingGroupOfCocoOrbit(fusionorb)`.

3.5.7 Size (for orbits of fusions)

▷ `Size(`*fusionorb*`)` (method)

returns the size of *fusionorb*.

3.5.8 AsList (for orbits of fusions)

▷ `AsList(`*fusionorb*`)` (method)

s expands the COCO2P-orbit object *fusionorb* into a list of fusions.

3.5.9 AsSet (for orbits of fusions)

▷ `AsSet(`*fusionorb*`)` (method)

expands the COCO2P-orbit object *fusionorb* into a set of fusions.

3.5.10 SubOrbitsOfCocoOrbit (for orbits of fusions)

▷ `SubOrbitsOfCocoOrbit(group, fusion)` (operation)

group is a subgroup of the underlying group of the orbit of fusions *fusionorb*. The given orbit splits into suborbits under this group. The function returns a list of these suborbits.

3.5.11 SubOrbitsWithInvariantPropertyOfCocoOrbit (for orbits of fusions)

▷ `SubOrbitsWithInvariantPropertyOfCocoOrbit(group, fusionorb, prop)` (operation)

prop is a function that takes a single fusion as argument and returns `true` or `false`. It has to be invariant under the partition-wise action of *group*. Note that the invariance is not checked by the function.

This function does the same as

```
Filtered(SubOrbitsOfCocoOrbit(group,fusionorb), x->prop(Representative(x)));
```

However, the former code is generally much less efficient than calling

```
SubOrbitsWithInvariantPropertyOfCocoOrbit(group,fusion,prop);
```

Chapter 4

Partially ordered sets

4.1 Introduction

COCO2P implements a data-type for partially ordered sets. The reason is, that for the posets of interest in COCO2P the test whether two elements are in order-relation is rather expensive, and COCO2P takes care to minimize the necessary tests. The other reason is, that this approach allows a nice and unified interface to XGAP for all kinds of posets that are introduced in COCO2P (i.e. posets of color graphs, posets of fusion orbits, lattices of fusions, lattices of closed sets, for now).

Like for combinatorial objects, COCO2P internally does not work directly with the elements of a poset, but instead uses indices into a list of elements (cf.). Only two functions refer directly to the elements: `CocoPosetByFunctions` (4.2.1) and `ElementsOfCocoPoset` (4.2.2). Therefore, in the following, we will identify the index to an element with the element.

4.2 General functions for COCO2P-posets

4.2.1 CocoPosetByFunctions

▷ `CocoPosetByFunctions(`*`elements, order, linpreorder`*`)` (function)

This is the main constructor for posets in COCO2P. All other constructors, behind the scenes, use this function.

`elements` is the underlying set of the poset.

`order` is a binary boolean function on *`elements`* that returns `true` on an input pair (x,y) is x is less than or equal y in the poset to be constructed. Otherwise it has to return `false`. The function *`order`* may be algorithmically difficult.

`linpreorder` is a binary boolean function that defines a linear preorder (reflexive, transitive, total relation) on *`elements`*, that extends the partial order relation defined by *`order`* such that the strict order of elements is preserved. That is, if y is strictly above x in *`order`*, then so it is in *`linpreorder`*.

The function *`linpreorder`* is used to speed up the computations of the successor-relation of the goal poset. It should be much quicker than *`order`* in order to really lead to a speedup. E.g., when computing a poset of sets, *`order`* may be the inclusion order, and *`linpreorder`* may be the function that compares cardinalities.

The function returns a COCO2P-poset object that encodes the poset defined by *`order`*.

4.2.2 ElementsOfCocoPoset

▷ `ElementsOfCocoPoset(poset)` (operation)

This function returns the list of elements of *poset*. Indices returned by other operations for posets, will be relative to this list.

4.2.3 Size (for COCO-posets)

▷ `Size(poset)` (method)

This function returns the number of elements of *poset*.

4.2.4 SuccessorsInCocoPoset

▷ `SuccessorsInCocoPoset(poset, i)` (operation)

This functions returns the successors of *i* in *poset*.

4.2.5 PredecessorsInCocoPoset

▷ `PredecessorsInCocoPoset(poset, i)` (operation)

This functions returns the predecessors of *i* in *poset*.

4.2.6 IdealInCocoPoset

▷ `IdealInCocoPoset(poset, set)` (operation)
▷ `IdealInCocoPoset(poset, i)` (operation)

This function returns the order ideal (a.k.a. downset) generated by *set* in *poset*.
In the second form, the principal order ideal generated by *i* in *poset* is returned.

4.2.7 FilterInCocoPoset

▷ `FilterInCocoPoset(poset, set)` (operation)
▷ `FilterInCocoPoset(poset, i)` (operation)

This function returns the order filter (a.k.a. upset) generated by *set* in *poset*.
In the second form, the principal order filter generated by *i* in *poset* is returned.

4.2.8 MinimalElementsInCocoPoset

▷ `MinimalElementsInCocoPoset(poset, set)` (operation)

This function returns the minimal elements of *set* in *poset*.

4.2.9 MaximalElementsInCocoPoset

▷ `MaximalElementsInCocoPoset(poset, set)` (operation)

This function returns the maximal elements of *set* in *poset*.

4.2.10 InducedCocoPoset

▷ `InducedCocoPoset(poset, set)` (function)

This function returns the subposet of *poset* that is induced by *set*

4.2.11 GraphicCocoPoset

▷ `GraphicCocoPoset(poset)` (operation)

This function creates a graphical representation of *poset* using XGAP.

4.3 Posets of color graphs

The class of color graphs of order n can be endowed with a preorder relation (i.e. a reflexive, transitive relation): We say that a color graph `cgr1` is sub color isomorphic to another color graph `cgr2` if there is a fusion color graph `cgr3` of `cgr2` that is color isomorphic to `cgr1`.

Restricted to a set of mutually non color isomorphic color graphs, the relation of sub color isomorphism induces a partial order. COCO2P is able to compute this induced order for lists of WL-stable color graphs.

4.3.1 ColorIsomorphicFusions

▷ `ColorIsomorphicFusions(cgr1, cgr2)` (function)

This function returns a list of all fusion orbits under the color automorphism group of *cgr1* whose representatives induce a color graph that is color isomorphic to *cgr2*.

At the moment this function is implemented only for WL-stable color graphs *cgr1* and *cgr2*.

4.3.2 SubColorIsomorphismPoset

▷ `SubColorIsomorphismPoset(cgrlist)` (function)

cgrlist is a list of WL-stable color graphs all of the same order and no two of them color isomorphic. The function returns a COCO2P-poset of *cgrlist* ordered by sub color isomorphism.

4.3.3 GraphicCocoPoset (for posets of color graphs)

▷ `GraphicCocoPoset(poset)` (method)

poset is a COCO2P-poset of colored graphs. This function creates a graphical representation of this poset. The labels of the nodes of the graphical poset correspond to the indices in the given poset.

The context-menu of each node gives additional information about the node. If for some node it is known whether the underlying color graph is surian or not, then this is made visible in the graphical poset. Nodes for which it is not known whether the cgr is Schurian, are represented by squares. Schurian nodes are represented by circles, and non-Schurian nodes are represented by diamonds.

This function is available only from XGAP.

Example

```
gap> lcgr:=AllAssociationSchemes(15);
[ AS(15,1), AS(15,2), AS(15,3), AS(15,4), AS(15,5), AS(15,6), AS(15,7),
  AS(15,8), AS(15,9), AS(15,10), AS(15,11), AS(15,12), AS(15,13), AS(15,14),
  AS(15,15), AS(15,16), AS(15,17), AS(15,18), AS(15,19), AS(15,20), AS(15,21),
  AS(15,22), AS(15,23), AS(15,24) ]
gap> Apply(lcgr, IsSchurian);
gap> pos:=SubColorIsomorphismPoset(lcgr);;
gap> GraphicCocoPoset(pos);
<graphic poset "Iso-poset of color graphs">
gap>
```

Chapter 5

Color Semirings

5.1 Introduction

Color semirings are an experimental feature that give an alternate interface to WL-stable color graphs, in the style of [Zie96] and [Zie05].

In the center stands the observation that the complexes (i.e., subsets of colors) of WL-stable color graphs can be endowed with a multiplication: Let $\Gamma = (V, C, f)$ be a WL-stable color graph with structure constants tensor T, and let M, N be subsets of the color set C. Then the product $M \cdot N$ is defined as the set of all colors k such that there exists $i \in M$, and $j \in N$ such that $T(i, j, k) > 0$. It is not hard to see that this operation is associative and that the set I of all reflexive colors is a neutral element. Moreover, this product-operation is distributive over the operation of union of complexes. Thus $(P(C), \cup, \cdot, \emptyset, I)$ forms a so-called semiring (cf. [Gol99], [Wik11]).

The color semiring of Γ acts naturally on the powerset $P(V)$ of the vertex set of Γ from the left and from the right. Let C be an element of the color semiring, and let M be a set of vertices of Γ. Then

$$C \cdot M := \{v \in V \mid \exists w \in M : f(v, w) \in C\},$$

$$M \cdot C := \{w \in V \mid \exists v \in M : f(v, w) \in C\}.$$

GAP has one operation symbold `+` for addition-like operations and one operation symbol `*` for multiplication-like operations. Thus in color semirings, the operation of union of complexes is denoted by `+`, and the operation of the product of complexes is denoted by `*`.

Since in COCO2P both, colors and vertices of a color graph are represented by positive integers, in order to distinguish complexes of colors and subsets of vertices, one of the two has to get its own type. The elements of color semirings (i.e., complexes of colors) all belong to the category `IsElementOfColorSemiring`. On the other hand, sets of vertices are simple sets of positive integers (no special category is created for them). In the GAP-output, complexes are denoted like `<[ a,b,c ]>`. The conversion of sets of colors to complexes is handled by the function `AsElementOfColorSemiring` (5.1.3), while the conversion of a complex to a set is done by the function `AsSet` (**Reference: AsSet**).

Example

```
gap> cgr:=JohnsonScheme(6,3);
<color graph of order 20 and rank 4>
gap> T:=StructureConstantsOfColorGraph(cgr);
<Tensor of order 4>
```

```
gap> sr:=ColorSemiring(cgr);
<ColorSemiring>
gap> s2:=AsElementOfColorSemiring(sr,[2]);
<[ 2 ]>
gap> s3:=AsElementOfColorSemiring(sr,[3]);
<[ 3 ]>
gap> s2*s3;
<[ 2, 3, 4 ]>
gap> ComplexProduct(T,[2],[3]);
[ 0, 4, 4, 9 ]
gap> 1*s2;
[ 2, 3, 4, 5, 6, 7, 11, 12, 13 ]
gap> Neighbors(cgr,1,2);
[ 2, 3, 4, 5, 6, 7, 11, 12, 13 ]
gap> Neighbors(cgr,1,3);
[ 8, 9, 10, 14, 15, 16, 17, 18, 19 ]
gap> 1*(s2+s3);
[ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 ]
```

Example

```
gap> g:=DihedralGroup(IsPermGroup,10);
Group([ (1,2,3,4,5), (2,5)(3,4) ])
gap> cgr:=ColorGraph(g, Combinations([1..5],2), OnSets,true);
<color graph of order 10 and rank 12>
gap> ColorMates(cgr);
(2,7)(3,10)(6,12)
gap> csr:=ColorSemiring(cgr);
<ColorSemiring>
gap> s2:=AsElementOfColorSemiring(csr,[2]);
<[ 2 ]>
gap> s3:=AsElementOfColorSemiring(csr,[3]);
<[ 3 ]>
gap> 1*(s2+s3);
[ 2, 3, 6, 7 ]
gap> Neighbors(cgr,1,[2,3]);
[ 2, 3, 6, 7 ]
gap> (s2+s3)*[2,3,6,7];
[ 1, 4, 5, 8, 10 ]
```

Many standard functions of GAP are applicable to color semirings, as a color semiring is just a structure, that is at the same time an additive magma with zero and a magma with one, such that multiplication and addition are associative and where the multiplication is distributive over the addition.

5.1.1 ColorSemiring

▷ `ColorSemiring(cgr)` (function)

cgr is a WL-stable color graph. The function returns an object, representing the color semiring of *cgr*

Example

```
gap> cgr:=JohnsonScheme(6,3);
<color graph of order 20 and rank 4>
gap> sr:=ColorSemiring(cgr);
<ColorSemiring>
gap> Elements(sr);
[ <[  ]>, <[ 1 ]>, <[ 1, 2 ]>, <[ 1, 2, 3 ]>, <[ 1, 2, 3, 4 ]>,
  <[ 1, 2, 4 ]>, <[ 1, 3 ]>, <[ 1, 3, 4 ]>, <[ 1, 4 ]>, <[ 2 ]>, <[ 2, 3 ]>,
  <[ 2, 3, 4 ]>, <[ 2, 4 ]>, <[ 3 ]>, <[ 3, 4 ]>, <[ 4 ]> ]
gap> List(last,AsSet);
[ [  ], [ 1 ], [ 1, 2 ], [ 1, 2, 3 ], [ 1, 2, 3, 4 ], [ 1, 2, 4 ], [ 1, 3 ],
  [ 1, 3, 4 ], [ 1, 4 ], [ 2 ], [ 2, 3 ], [ 2, 3, 4 ], [ 2, 4 ], [ 3 ],
  [ 3, 4 ], [ 4 ] ]
```

5.1.2 GeneratorsOfColorSemiring

▷ `GeneratorsOfColorSemiring(`*csr*`)` (attribute)

This function returns a list of additive generators of the color semiring *csr*.

Example

```
gap> cgr:=JohnsonScheme(6,3);
<color graph of order 20 and rank 4>
gap> sr:=ColorSemiring(cgr);
<ColorSemiring>
gap> gens:=GeneratorsOfColorSemiring(sr);
[ <[ 1 ]>, <[ 2 ]>, <[ 3 ]>, <[ 4 ]> ]
```

5.1.3 AsElementOfColorSemiring

▷ `AsElementOfColorSemiring(`*csr*`, `*cset*`)` (function)

This function takes as input a color semiring *csr* and a set of colors *cset*. It returns the element of *csr* that corresponds to *cset*.

Example

```
gap> cgr:=JohnsonScheme(6,3);
<color graph of order 20 and rank 4>
gap> sr:=ColorSemiring(cgr);
<ColorSemiring>
gap> s2:=AsElementOfColorSemiring(sr,[2]);
<[ 2 ]>
gap> s3:=AsElementOfColorSemiring(sr,[3]);
<[ 3 ]>
gap> s2*s3;
<[ 2, 3, 4 ]>
```

References

[BIK89] A. E. Brouwer, A. V. Ivanov, and M. H. Klin. Some new strongly regular graphs. *Combinatorica*, 9(4):339–344, 1989. 7

[Gol99] Jonathan S. Golan. *Semirings and their applications.* Kluwer Academic Publishers, Dordrecht, 1999. 39

[Iva89] A. V. Ivanov. Non-rank-3 strongly regular graphs with the 5-vertex condition. *Combinatorica*, 9(3):255–260, 1989. 7

[Iva94] A. V. Ivanov. Two families of strongly regular graphs with the 4-vertex condition. *Discrete Math.*, 127(1-3):221–242, 1994. 8

[Wei76] B. Weisfeiler. *On construction and identification of graphs*, volume 558 of *Lecture Notes in Math.* Springer, Berlin, 1976. 4

[Wik11] Wikipedia. Semiring — wikipedia, the free encyclopedia, 2011. [Online; accessed 5-April-2011]. 39

[WL68] B. J. Weisfeiler and A. A. Leman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno - Technicheskaja Informatsia*, 9(Seria 2):12–16, 1968. (Russian). 4

[Zie96] Paul-Hermann Zieschang. *An algebraic approach to association schemes*, volume 1628 of *Lecture Notes in Mathematics*. Springer, Berlin, 1996. 39

[Zie05] Paul-Hermann Zieschang. *Theory of association schemes*. Monographs in Mathematics. Springer, Berlin, 2005. 39

Index