GAP 4 Package genss

generic Schreier-Sims

1.3

September 2009

Max Neunhöffer
Felix Noeske

Max Neunhöffer — Email: neunhoef@mcs.st-and.ac.uk
— Homepage: http://www-groups.mcs.st-and.ac.uk/~neunhoef
— Address: School of Mathematics and Statistics Mathematical Institute University of St Andrews North Haugh St Andrews, Fife KY16 9SS Scotland, UK

Felix Noeske — Email: felix.noeske@math.rwth-aachen.de
— Homepage: http://www.math.rwth-aachen.de/~Felix.Noeske
— Address: Lehrstuhl D für Mathematik, RWTH Aachen, Templergraben 64, 52062 Aachen, Germany

Copyright

Contents

Chapter 1

Introduction

1.1 Philosophy

This package is about the Schreier-Sims algorithm and base and strong generating sets for arbitrary finite groups.

1.2 Overview over this manual

Chapter 2 describes the installation of this package. Chapter 3 desribes the core functionality of the package, namely the computation and usage of stabiliser chains. Chapter 4 describes some methods using backtrack search. This part of the package is not yet fully developed and is subject to change in later versions. Finally, Chapter 5 shows instructive examples for the usage of this package. As of now, the number of shown examples is zero.

Chapter 2

Installation of the genss-Package

To install this package just extract the package's archive file to the GAP `pkg` directory.

By default the genss package is not automatically loaded by GAP when it is installed. You must load the package with `LoadPackage("genss");` before its functions become available.

Please, send us an e-mail if you have any questions, remarks, suggestions, etc. concerning this package. Also, I would like to hear about applications of this package.

Max Neunhöffer and Felix Noeske

Chapter 3

Stabiliser Chains

This chapter describes the core functionality of the package. It mainly covers how to use genss to compute stabiliser chains for GAP groups and use them to do sifting.

3.1 Computing stabiliser chains

The main tool to compute a stabiliser chain is the following operation. It has many options and can be customised in a very flexible way.

3.1.1 StabilizerChain

◊ `StabilizerChain(G[, opt])` (operation)

Returns: a stabiliser chain object or `fail`

This operation computes a stabiliser chain for the group *G* using a randomised Schreier-Sims algorithm. The second argument *opt* is an optional options record. See Section 3.2 below for an explanation of the possible components.

Note that this is a Monte Carlo algorithm in most cases, so there is a small error probability. However, the only error possible is that some of the subgroups in the stabiliser chain are proper subgroups of the actual point stabilisers. So the resulting group order is always a divisor of the actual group order and if the two are equal, then the stabiliser chain is proved to be correct. In particular, if the group object *G* for some reason already knows the group order, then this procedure always returns a correct and proven stabiliser chain for *G*.

3.2 Options for the computation of stabiliser chains

The options record for the `StabilizerChain` (3.1.1) can contain the following components, which are used to control the behaviour of the computation of a stabiliser chain for the group *G*. Note that for most of these components there are default values to be found in the global `GENSS` record. You can change these defaults there if you want but you should know what you are doing. An explicitly given value in the options record always takes precedence over the default value.

`Base` This component can either be bound to an existing stabilizer chain object or to a list of points. In both cases this indicates that the base of the stabilizer chain or the list of points respectively are known to be a base for the group *G*. In the first case the corresponding action functions are taken from the stabiliser chain, in the second case one should usually bind the component `BaseOps` to a list of equal length containing the action functions corresponding to the base points.

`BaseOps` If the `Base` component is bound to a list of points the `BaseOps` component must be a list of equal length containing the corresponding action functions. If the `BaseOps` component is unbound, a list with identical entries `OnLines` is used in projective mode and `OnPoints` in non-projective mode (see component `Projective` below).

`Cand` The `Cand` component can be bound to a record `r`which contains candidates for base points in the following way. The `r.points` component contains the list of points, the `r.ops` component contains a list of equal length with the corresponding action functions. The points and actions specified in the `Cand` component are tried as possible base points for *G* (and its stabilisers) first before other points are guessed (see `FindBasePointCandidates` (3.3.1)). If a point is fixed under all generators it is not used, unless the component `Reduced` is explicitly set to `false` (see below). If the component `StrictlyUseCandidates` is `false` (the default, see below), the algorithm tries to use other points of an already found orbit before considering the next candidate specified by `Cand`. This is usually sensible because for an already enumerated orbit we have a natural bound on the length of the suborbits for the point stabiliser in this orbit.

`DeterministicVerification` Set this component to `true` to switch on a deterministic verification routine after the randomised Schreier-Sims procedure. This is not yet implemented.

`ErrorBound` Set this component to a rational number between 0 and 1. It will be an upper bound for the error probability. That is, the error probability of the Monte Carlo verification at the end will be less than this rational number. This component overrides everything you specify in the `random` or `VerifyElements` components.

`FailInsteadOfError` If no short enough orbit is found during the computation, the procedure stops with an error message. If you would rather like it to return `fail` then set this component to `true`. This option can be used to try an stabiliser chain computation automatically and give up before you run out of memory.

`ImmediateVerificationElements` Whenever the randomised Schreier-Sims procedure has first computed generators for a stabiliser in the chain and has computed a stabiliser chain for that stabiliser recursively, an immediate verification is done. This is to spot early on that the group found is in fact a proper subgroup of the stabiliser. This verification is done by creating a few more random elements of that stabiliser and sifting them through the newly created stabiliser chain. Each such element has a chance of at least $1/2$ to spot this. The number of random elements used is stored in the component `ImmediateVerificationElements`.

`InitialHashSize` Set this component to the initial tree hash size for orbit computations in the stabiliser chain.

`IsOne` The default for this computation is the `IsOne` (**Reference: IsOne**) operation in the GAP library. Whenever in the stabiliser chain computation it has to be tested whether or not a group element is equal to the identity, the function stored in the `IsOne` component is called. The rationale behind this is that you can compute a stabiliser chain for a factor group of the group object *G*. For example, if you set the `IsOne` component to `GENSS_IsOneProjective` (3.2.1) for a matrix group *G*, scalar multiples of the identity are considered to be equal to the identity. You will have to specify the base points explicitly using the `Cand` and `StrictlyUseCandidates` component (see above and below) to only use actions having the normal subgroup in its kernel. A shortcut for computing stabiliser chains of projective groups (matrix group modulo scalars) is to set the `Projective` component (see below) and switch to projective mode.

`LimitShortOrbCandidates` The integer value of this component limits the number of candidates considered for the finding of short orbits. See the `TryShortOrbit` and `TryBirthdayParadox` components.

`NrRandElsBirthdayParadox` The method using the birthday paradox to find short orbits uses at most as many random group elements to estimate the orbit size as this component says. See the `TryBirthdayParadox` component.

`NumberPrevOrbitPoints` After an orbit for the stabiliser chain has been enumerated, the randomised Schreier-Sims method first tries `NumberPrevOrbitPoints` from this orbit as next base points. Note that this is not done if the `StrictlyUseCandidates` component is set to `true`.

`OrbitLengthLimit` This component is an absolute upper bound for the length of all orbits in the stabiliser chain. If an orbit enumeration reaches this limit, the stabiliser chain computation is aborted.

`OrbitLimitBirthdayParadox` During the method to find short orbits using the birthday paradox (see component `TryBirthdayParadox`) only orbits whose final estimated length is less than `OrbitLimitBirthdayParadox` are used.

`OrbitLimitImmediatelyTake` During the method to find short orbits using the birthday paradox (see component `TryBirthdayParadox`) an orbit is immediately used if its currently estimated length is less than `OrbitLimitImmediatelyTake`, even if the estimate is not yet very reliable.

`Projective` Set this component to `true` if you want to compute a stabiliser chain for a projective group given as a matrix group. Elements which are scalar multiples of each other will be considered to be equal. This is achieved by only considering projective actions. Note that in this case a known size of the group object cannot be used, since this size is the order of the matrix group!

`random` The `random` component is there as a compatibility option. It behaves exactly as for the stabiliser chain methods in the GAP library. It must be set to a number between 0 and 1000 indicating a lower bound for the probability of a correct answer, where the value a means $a/10\%$. Note that currently 1000 is not yet implemented since there is no working deterministic verification routine.

`RandomElmFunc` If this component is bound then it must be bound to a function taking no arguments and returning uniformly distributed random elements in the group for which the stabiliser chain is to be computed. All random elements used for the stabiliser chain will then be created by calling this function.

`RandomStabGens` This component contains the number of random stabiliser elements that are generated initially to generate a new stabiliser in the chain.

`Reduced` If this component is bound to `true`, then no orbits of length 1 are allowed in the stabiliser chain. That is, no points are taken as base points that are fixed under all generators of the current stabiliser. Set this component to `false` to allow for orbits of length 1, for example if you want the stabiliser chain to run through a prescribed base.

`Report` The number in the `Report` component is taken as the `Report` component in all orbit enumerations. That is, every `Report` newly found elements in the orbit a message is printed saying how far the computation has gone.

`ShortOrbitsInitialLimit` See the `TryShortOrbit` component.

`ShortOrbitsNrRandoms` See the `TryShortOrbit` component.

`ShortOrbitsOrbLimit` See the `TryShortOrbit` component.

`Size` If the `Size` component is set to a positive integer it is taken as the size of the group G. This information allows to verify the stabiliser chain simply by looking at its size. If the group object knows its size already (and the `Projective` component was not set to `true`), then the stored size of the group is automatically taken into account, such that one does not have to use this option.

`StabGenAddSlots` The value of the `StabGenAddSlots` component is directly handed over to the product replacer object which is used to generate random elements to find stabiliser generators.

`StabGenMaxDepth` The value of the `StabGenMaxDepth` component is directly handed over to the product replacer object which is used to generate random elements to find stabiliser generators.

`StabGenScrambleFactor` The value of the `StabGenScrambleFactor` component is directly handed over to the product replacer object which is used to generate random elements to find stabiliser generators.

`StabGenScramble` The value of the `StabGenScramble` component is directly handed over to the product replacer object which is used to generate random elements to find stabiliser generators.

`StrictlyUseCandidates` If this component is set to `true` (default is `false`) then only the given candidate points are taken as possible base points. In particular, the procedure does not take additional points of the previous orbit as candidates for base points (see component `NumberPrevOrbitPoints`). Use this option in combination to `Reduced` set to `false` to enforce a certain known base.

`TryBirthdayParadox` The method to try to find short orbits using the birthday paradox is used up to `TryBirthdayParadox` times for each new base point. This method uses the Murray/O'Brien heuristics to find candidates for short orbits and then uses statistics using the birthday paradox to estimate the orbit lengths. As soon as a point is found whose orbit is either estimated to be smaller than `OrbitLimitBirthdayParadox` with a solid statistical estimate or is estimated to be smaller than `OrbitLimitImmediatelyTake` with a weak statistical estimate, this point is taken as the next base point.

`TryShortOrbit` The method to try to find short orbits using the standard Murray/O'Brien heuristics is used up to `TryShortOrbit` times for each new base point. This method uses the heuristics to find candicates for short orbits using `ShortOrbitsNrRandoms` random group elements.

It then enumerates all these orbits up to the limit `ShortOrbitsInitialLimit`. If any of them closes the corresponding candidate is taken as the next base point. Otherwise half of the points are thrown away and the limit is doubled. This goes on until either an orbit closes or the limit grows over `ShortOrbitsOrbLimit`.

`VerifyElements` This component can be used to set it to the number of random elements that are used in the end to verify the stabiliser chain statistically. Usually the user specifies the component `ErrorBound` instead and `VerifyElements` is then computed automatically from that. However, if no `ErrorBound` is given, the `VerifyElements` component takes precedence over the `random` component.

`VeryShortOrbLimit` The very first method tried to find the next base point is to enumerate the orbit of the first and the last basis vector and of one random vector up to the limit `VeryShortOrbLimit`. If the orbit closes before this limit is reached, the corresponding vector is immediately taken.

3.2.1 GENSS_IsOneProjective

◊ `GENSS_IsOneProjective(`*x*`)` (function)

Returns: `true` or `false`

This function tests whether or not the matrix *x* is a scalar multiple of the identity matrix. This function is useful for projective action.

3.3 How base points are chosen

This section explains some internal details of how base points are chosen during a stabiliser chain computations. As a user, you can probably safely skip this section and ignore it altogether. However, in situations where thes stabiliser chain computation is more difficult (for example if it is difficult to find short orbits), then it can be helpful to know about these details.

Whenever the stabiliser chain computation needs to set up a new layer in the stabiliser chain it needs a new base point. The first method it tries is to take a point in the previous orbit one layer up, since for these points we have a natural upper limit for the orbit length, namely the orbit length in the layer above. If this does not work (either there is no higher layer or more than the first `NumberPrevOrbitPoints` (see stabiliser chain options in Section 3.2) in that orbit are fixed by the current group or `StrictlyUseCandidates` is `true`), it is checked whether or not there is another known candidate for a base point.

Note that if the user supplies candidates for the base points and operations (see component `Cand` in the stabiliser chain options in Section 3.2), then it is entirely possible that all base points come from these candidates and the mechanisms described in this sections are not used at all.

However, if the procedure runs out of base points, it needs a way to find new candidates. This is done using the following operation:

3.3.1 FindBasePointCandidates

◊ `FindBasePointCandidates(`*g*`,` *opt*`,` *i*`,` *S*`)` (operation)

Returns: a `Cand` record

This operation returns base point candidates in the form of a record as for the `Cand` option for stabiliser chain computations (see Section 3.2).

There are various methods installed to this end which all might fail and call `TryNextMethod();`. We do not document the details of these methods here but only give an overview. For permutation groups the choice of candidates is very straightforward, one simply takes a few integers with the usual action `OnPoints`. For matrix group finding a reasonably short orbit is more difficult. The system first handles the case of a scalar group which is easy. Then it hopes to find a "very short orbit" defined by the component `VeryShortOrbLimit` in the stabiliser chain computation options. If this fails the birthday paradoxon method is used to find an estimate for a reasonably short orbit amoung candidates coming from the Murray and O'Brien heuristics. If this fails the same heuristics are used but various orbits are enumerated up to a certain point decreasing the number of orbits as the limit goes up. If all fails some of the candidates from the heuristics are simply tried with brute force. The whole computation can fail if some upper orbit length limit is reached (see component `OrbitLengthLimit` in the stabiliser chain computation options).

3.4 Using stabiliser chains

The most important thing one can do with a stabiliser chain is sifting. This is done with one of the next to operations:

3.4.1 SiftGroupElement

◊ `SiftGroupElement(`*`S, el`*`)` (operation)

Returns: a record

The first argument *S* must be a stabiliser chain object and the second argument *el* a group element (not necessarily contained in the group described by *S*). The result is a record describing the result of the sifting process. The component `rem` contains the remainder of the sifting process. If *el* is contained in the group described by *S*, then the remainder is equal to the identity. Note that if the `IsOne`-component of the options record for the stabiliser chain *S* is different from the `IsOne` (**Reference: IsOne**) operation then the `rem` component is equal to the identity according to that test. The result of this test (`true` or `false`) is stored in the component `isone` of the resulting record. This means, that this component indicates whether or not the sifting was successful. The component `S` is bound to the stabiliser chain object corresponding to the layer in which the sifting stopped. If it ran through the whole chain this component is bound to `false`. The component `preS` is always bound to the previous layer, which is the lowest layer if the sifting was successful.

3.4.2 SiftGroupElementSLP

◊ `SiftGroupElementSLP(`*`S, el`*`)` (operation)

Returns: a record

This operation behaves exactly as `SiftGroupElement` (3.4.1) except that in the successful case the component `slp` of the resulting record is additionally bound to a straight line program which expresses the element *el* in terms of the strong generators of the stabiliser chain (see `StrongGenerators` (3.4.3)).

3.4.3 StrongGenerators

◊ `StrongGenerators(`*`S`*`)` (operation)

Returns: a list of group elements

This operation returns the strong generators of the stabiliser chain `S`. This means that each stabiliser in the stabiliser chain is generated by the subset of the set of strong generators which fix the corresponding points. Note that each layer of the stabiliser chain uses some subset of these strong generators as generators for the orbit object of that layer.

3.4.4 NrStrongGenerators

◊ `NrStrongGenerators(S)` (operation)

Returns: a positive integer

This operation returns the number of strong generators of the stabiliser chain `S` (see `StrongGenerators` (3.4.3)).

3.4.5 BaseStabilizerChain

◊ `BaseStabilizerChain(S)` (operation)

Returns: a record

This operation returns the base of the stabiliser chain `S` in form of a record, which can be used as the `Cand` component for a stabiliser chain computation. That is, two components are bound, the `points` component is a list of base points and the `ops` component is a corresponding list of action functions.

3.4.6 Size

◊ `Size(S)` (operation)

Returns: a positive integer

This operation returns the size (i.e. order) of the group described by the stabiliser chain `S`. This is simply the product of the lengths of the orbits in the chain.

3.4.7 Random

◊ `Random(S)` (operation)

Returns: a group element

This operation can be called with a stabiliser chain object `S` or with a group object, if this group object has a stored stabiliser chain (see `SetStabilizerChain` (3.4.11)). The method will randomly choose transversal elements and thus produce a uniformly distributed random element of the group.

3.4.8 \in

◊ `\in(x, S)` (operation)

Returns: `true` or `false`

This operation tests whether or not the group element x lies in the group described by the stabiliser chain `S` by sifting (see `SiftGroupElement` (3.4.1)). The argument `S` can also be a group object with a stored stabiliser chain (see `SetStabilizerChain` (3.4.11)). Note that this operation can be called with the `in` keyword using infix notation.

3.4.9 IsProved

◊ `IsProved(S)` (operation)

Returns: `true` or `false`

This operation returns whether or not the stabiliser chain *S* is proved to be correct. If it has only been verified by randomised methods, `false` is returned. At the time of this writing the only possible deterministic verification is if the size of the group is known before the stabiliser chain computation begins.

3.4.10 GroupIteratorByStabilizerChain

◊ `GroupIteratorByStabilizerChain(S)` (operation)

Returns: an iterator

This operation returns an iterator object which runs through the elements of the group described by the stabiliser chain object *S*. The usual operations `NextIterator` (**Reference: NextIterator**) and `IsDoneIterator` (**Reference: IsDoneIterator**) as well as the `for` loop construction can be used with this object. The iterator is implemented using the stored transversals in the Schreier trees of the stabiliser chain.

3.4.11 SetStabilizerChain

◊ `SetStabilizerChain(g, S)` (operation)

Returns: nothing

Once the user is convinced that the stabiliser chain *S* describes the group *g* correctly, he can call this operation to store the stabiliser chain together with the group object. From then on, additional methods using the stabiliser chain (for example `Size` (3.4.6), `Random` (3.4.7) and `\in` (3.4.8) above) become applicable for the group object. Note that if a stabiliser chain is known to be correct (for example if the group knew its size beforehand), then the stabiliser chain is stored with the group automatically when it is constructed, which makes the explicit storing of the stabiliser chain unnecessary.

The stored stabiliser chain of a group object can be used using `StoredStabilizerChain` (3.4.12).

3.4.12 StoredStabilizerChain

◊ `StoredStabilizerChain(g)` (attribute)

Returns: a stabiliser chain

This attribute for a group object *g* contains a stored stabiliser chain for the group. See `SetStabilizerChain` (3.4.11) for details.

3.4.13 StabChainOp

◊ `StabChainOp(p, S)` (method)

Returns: a GAP stabiliser chain

This method computes a standard GAP library stabiliser chain for the permutation group *p* using the fact that *S* is a known correct stabiliser chain for *p*. If all base points in *S* are positive integers and all actions are equal to `OnPoints`, then the same base points are taken for the new stabiliser chain.

3.4.14 SiftBaseImage

◊ `SiftBaseImage(S, l)` (operation)

Returns: `true` or `false`

This operation sifts an image *l* of the base points of the stabiliser chain *S*. This means that the elements of the list *l* must be images of the base points under the actions in the various layers of the stabiliser chain. The sifting procedure using the orbits and Schreier trees in the stabiliser chain decides if this base image is one for a group element of the group described by *S* and returns `true` or `false` accordingly.

This operation is mostly used internally.

3.4.15 SLPChainStabilizerChain

◊ `SLPChainStabilizerChain(S, gens)` (operation)

Returns: a record

This operation assumes that *S* is a stabiliser chain that correctly describes the group generated by the generators *gens*. It returns a list of straight line programs expressing successively the stabilisers in the chain, each in terms of the generators of the previous, the first in terms of *gens*. This list is stored in the component `slps` of the resulting record. The sizes of the groups in the chain are stored in the component `sizes` of the resulting record.

The operations, functions and methods described below use stabiliser chains internally:

3.4.16 GroupHomomorphismByImagesNCStabilizerChain

◊ `GroupHomomorphismByImagesNCStabilizerChain(g, h, images, opt1, opt2)` (function)

Returns: a group homomorphism

This function creates a group homomorphism object from the group *g* into the group object *h*, mapping the generators of the group *g* to the elements *images* which must lie in *h*. This mapping must be a group homomorphism, note that this is not checked!

The homomorphism is computed by computing stabiliser chains on both sides such that elements can be mapped in both directions simply be sifting and expressing them in terms of the strong generators. This is where the two arguments *opts1* and *opts2* come into play. The former is used as the options record for the stabiliser computation in *g* and the latter for the one in the group generated by *images*.

3.4.17 FindShortGeneratorsOfSubgroup

◊ `FindShortGeneratorsOfSubgroup(g, u)` (method)

Returns: a record

This is an additional method for matrix or permutation groups implementing the operation `FindShortGeneratorsOfSubgroup` from the orb package using stabiliser chains. Both arguments must be groups and *u* must be a subgroup of *g*. The resulting record contains two components `gens` and `slp`, where the first is a list of generators for the group *u* and the second is a straight line program expressing `gens` in terms of the generators of *g*. This operation aims to find short words in the generators of *g* to use as generators for *u*.

3.4.18 Stab

◊ `Stab(`*g*`, `*x*`, `*op*`[, `*opt*`])` (operation)

Returns: a record or `fail`

This operation aims to compute the point stabiliser of the group *g* acting via the action function *op* of the point *x*. The optional last argument is an options record. The general approach of this procedure is to go back and forth between enumerating a part of the orbit and trying to produce random elements in the stabiliser using the already enumerated part of the orbit. Random elements in the stabiliser are produced by using product replacement in *g* to produce random elements of *g* and then using the Schreier tree of the orbit to map them back into the stabiliser. If this works, the resulting random elements are distributed uniformly in the point stabiliser.

This routine is a Monte Carlo procedure. If sufficiently many random elements of the stabiliser have been produced and did not increase its size, the program concludes that the whole stabiliser is found and returns a record describing it. Otherwise it returns `fail` after some time.

The resulting record has the stabiliser in the component `stab`, its size estimate in the component `size`, a stabiliser chain for `stab` in the component `stabilizerchain` and a boolean value in the component `proof` indicating whether or not the result is certain.

We do not document all possible options in the options record here, since we want to allow for the possibility to change these in later versions. The most important component in the options record is the component `ErrorBound` which must be bound to a rational number between 0 and 1 and which is an upper bound for the error probability.

Please note again that two types of errors can occur in this program: The first is that the correct point stabiliser is not found but only a proper subgroup of it. The second is that the stabiliser chain computation to estimate its size went wrong and returns an incorrect stabiliser chain.

Chapter 4

Backtrack search methods

This chapter describes the methods for backtrack search in the genss package. Note that the code in this area is not yet very stable and is almost certainly going to change in subsequent versions of this package. This might also concern the interfaces and calling conventions.

4.1 Setwise stabilisers

4.1.1 SetwiseStabilizer

◊ `SetwiseStabilizer(`*G, op, M*`)` (operation)

Returns: a record

This operation computes the setwise stabiliser of the set *M*. So *G* must be a group acting on some set Ω, this action is given by the action function *op*. The set *M* must consist of elements Ω. The result is a record with the components `setstab` containing the setwise stabiliser and `S` containing a stabiliser chain for it.

This operation uses backtrack search in a specially crafted stabiliser chain for *G* doing not much intelligent pruning of the search tree, so expect possible long delays!

4.1.2 SetwiseStabilizerPartitionBacktrack

◊ `SetwiseStabilizerPartitionBacktrack(`*G, op, M*`)` (operation)

Returns: a record

This operation computes the setwise stabiliser of the set *M*. So *G* must be a group acting on some set Ω, this action is given by the action function *op*. The set *M* must consist of elements Ω. The result is a record with the components `setstab` containing the setwise stabiliser and `S` containing a stabiliser chain for it.

This operation uses backtrack search in a specially crafted stabiliser chain for *G*. It does some ideas coming from partition backtrack but does not (yet) implement a full featured partition backtrack, so expect possible longish delays!

4.2 Generic backtrack search

4.2.1 BacktrackSearchStabilizerChainElement

◊ `BacktrackSearchStabilizerChainElement(S, P, g, pruner)` (operation)

Returns: `fail` or a group element

Let G be the group described by the stabiliser chain `S`. The group element `g` must be some element in an overgroup $\hat{G}$of G such that the function `P` described below is defined on the whole of $\hat{G}$

This operation implements a generic backtrack search in the coset G`g` looking for an element $x\ inG$ such that `P`$(x$`g`$)$ is `true` where `P` is a function on $\hat{G}$taking values `true` and `false`. The operation returns the group element x if one is found or `fail` if none was found.

The search tree is given by the stabiliser chain, each node corresponds to a right coset of one of the stabilisers in the chain. The leaves correspond to right cosets of the identity group, i.e. to group elements in G`g`

To make this backtrack search efficient some pruning of the search tree has to be done. To this end there is the fourth argument `pruner` which can either be `false` (in which case no pruning at all happens) or a GAP function taking 5 arguments and returning either `true` or `false`. The function `pruner` is called for every node in the search tree before the backtrack search descents into the subtrees. If the `pruner` function returns `false`, the complete subtree starting at the current node is pruned and no further search is performed there. If the result is `true` (or `pruner` was equal to `false` altogether) then the subtree starting at the current node is searched recursively. Obviously, the `pruner` function needs to know the current position in the search tree, which it is told by its arguments.

Each node in the search tree corresponds to a coset of some stabiliser of the stabiliser chain in its previous one. To set up some notation, let $G = S_0 > S_1 > S_2 > \cdots > S_m > S_{m+1} = \{1\}$ be the stabiliser chain and let $O_1, O_2, \ldots, O_m$ be the basic orbits. Then for the node corresponding to the coset $S_i t$`g` for $i \geq 1$ and some transversal element t contained in S_{i-1} the arguments with which the `pruner` function is called are the following: The first argument is the stabiliser chain object corresponding to S_{i-1}. The second argument is the index of the element in O_i corresponding to the transversal element t. The third argument is the group element t`g` and the fourth argument is equal to the actual transversal element t. The fifth argument is a word in the generators used to enumerate O_i expressing t, the word comes as a list of integers which are the generator numbers.

4.2.2 BacktrackSearchStabilizerChainSubgroup

◊ `BacktrackSearchStabilizerChainSubgroup(S, P, pruner)` (operation)

Returns: `fail` or a stabiliser chain

Let G be the group described by the stabiliser chain `S`. This operation implements a generic backtrack search in the stabiliser chain `S` looking for the subgroup H of the group G described by `S` of all elements x for which `P`(x) is `true`, where `P` is a function on G taking values `true` or `false`. Note that of course `P` must be such that H is actually a subgroup! The operation returns a stabiliser chain describing the group H.

The search tree is given by the stabiliser chain, each node corresponds to a right coset of one of the stabilisers in the chain. The leaves correspond to right cosets of the identity group, i.e. to group elements in G

To make this backtrack search efficient some pruning of the search tree has to be done. To this end there is the fourth argument `pruner` which can either be `false` (in which case no pruning at all happens) or a GAP function taking 5 arguments and returning either `true` or `false`. The function *`pruner`* is called for every node in the search tree before the backtrack search descents into the subtrees. If the *`pruner`* function returns `false`, the complete subtree starting at the current node is pruned and no further search is performed there. If the result is `true` (or *`pruner`* was equal to `false` altogether) then the subtree starting at the current node is searched recursively. Obviously, the *`pruner`* function needs to know the current position in the search tree, which it is told by its arguments.

Each node in the search tree corresponds to a coset of some stabiliser of the stabiliser chain in its previous one. To set up some notation, let $G = S_0 > S_1 > S_2 > \cdots > S_m > S_{m+1} = \{1\}$ be the stabiliser chain and let $O_1, O_2, \ldots, O_m$ be the basic orbits. Then for the node corresponding to the coset $S_i t g$ for $i \geq 1$ and some transversal element t contained in S_{i-1} the arguments with which the *`pruner`* function is called are the following: The first argument is the stabiliser chain object corresponding to S_{i-1}. The second argument is the index of the element in O_i corresponding to the transversal element t. The third and fourth arguments are the transversal element t. The fifth argument is a word in the generators used to enumerate O_i expressing t, the word comes as a list of integers which are the generator numbers.

Chapter 5

Examples

Here comes text.

References

Index