GAP 4 Package orb

orb — Methods to enumerate Orbits

Version 3.3

September 2009

Jürgen Müller
Max Neunhöffer
Felix Noeske

Jürgen Müller — Email: juergen.mueller@math.rwth-aachen.de
— Homepage: http://www.math.rwth-aachen.de/~Juergen.Mueller
— Address: Lehrstuhl D für Mathematik, RWTH Aachen, Templergraben 64, 52062 Aachen, Germany

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 Motivation for this package

This package is about orbit enumeration. It bundles fundamental algorithms for orbit enumeration as well as more sophisticated special-purpose algorithms for very large orbits.

The fundamental methods are basically an alternative implementation to the orbit algorithms in the GAP library. We tried to make them more flexible and more efficient at the same time, therefore backwards compatibility with respect to the user interface had to be given up. In addition, more information about how an orbit was produced is retained and is available for further usage. These orbit enumeration algorithms build on even more fundamental code for hash tables.

The higher level algorithms basically implement the idea to enumerate an orbit "by suborbits" with respect to one or more subgroups. While these orbit-by-suborbit algorithms are much more efficient in many cases, they very often need careful and sometimes difficult preparations by the user. They are definitely not intended to be "push-the-button-tools" but require a considerable amount of knowledge from the "pilot".

Quite a bit of the code in this package consists in fact of interactive tools to enable users to prepare the data for the orbit-by-suborbit algorithms to work.

1.2 Overview over this manual

Chapter 2 describes the installation of this package. Chapter 3 describes our reimplementation of the basic orbit algorithm. Chapter 4 describes our toolbox for hash tables, Chapter 5 explains caching data structures, whereas Chapter 8 describes our implementation of AVL trees. Chapter 6 covers tools to use random methods in groups. Chapter 7 describes a lot of tools to search in groups and orbits. These techniques are basically intended to provide the data structures necessary to run the code described in Chapter 9 to use the orbit-by-suborbit algorithms. Currently, Chapter 10 is an empty placeholder. In some future version of this package it will contain a description of code which helps users to find nice quotients of modules which is also needed for the orbit-by-suborbit algorithms. However, since the interface to this code is not yet stable, we chose not to document it as of now, in particular because it relies on other not yet published packages as of the time of this writing. Finally, Chapter 11 shows an instructive examples for the more sophisticated usage of this package.

Chapter 2

Installation of the orb-Package

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

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

As of version 3.0, the orb package has a GAP kernel component which must be compiled. This component does not actually contain new functionality but will improve the performance of AVL trees and hash tables significantly since many core routines are implemented in the C language at kernel level.

To compile the C part of the package do (in the `pkg` directory)

```
cd orb
./configure [path]
```

where "path" is a path to the main GAP root directory (if not given the default "`../..`" is assumed).

Afterwards call "`make`" to compile a binary file.

If you installed GAP on several architectures, you must execute this configure/make step on each of the architectures immediately after configuring GAP itself on this architecture.

If you installed the package in another "`pkg`" directory than the standard "`pkg`" directory in your GAP 4 installation, then you have to add the path to the directory containing your "`pkg`" directory to GAP's list of directories. This can be done by starting GAP with the "`-l`" command line option followed by the name of the directory and a semicolon. Then your directory is prepended to the list of directories searched. Otherwise the package is not found by GAP. Of course, you can add this option to your GAP startup script.

2.1 Static linking

This might be interesting for M$ Windows users, as dynamic loading of binary modules does not always work there. You can also create a new statically linked "`gap`" binary as follows:

Go into the main GAP directory and then into `bin/BINDIR`. Here `BINDIR` means the directory containing the "`gap`" executable after compiling "`gap`". This directory also contains the GAP compiler script "`gac`". Assuming orb in the standard location you can then say

```
./gac -o gap-static -p "-DORBSTATIC" -P "-static" ../../pkg/orb/src/orb.c
```

Then copy your "`gap`" start script to, say, "`gaps`" and change the references to the GAP binary to "`gap-static`".

Note that you have to replace `BINDIR` by the name containing the "`gap`" executable after compiling GAP as above. If you have installed the package in a different place than the standard, you have to replace "`../..`" in the above command by the path to the directory containing the "`pkg`" directory into which you installed orb. If you want to install more than one package with a C-part like this package, you can still create a statically linked GAP executable by combining all the compile and link options and all the .c files as in the ./gac command above. For the orb package, you have to add

```
-DORBSTATIC
```

to the string of the -p option and the file

```
../../pkg/orb/src/orb.c
```

somewhere on the command line. As above, "`../..`" and "`BINDIR`" have to be replaced if you installed in a non-standard location.

2.2 Recompiling the documentation

Recompiling the documentation is possible by the command "`gap makedoc.g`" in the `orb` directory. But this should not be necessary.

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.

Jürgen Müller, Max Neunhöffer and Felix Noeske

Chapter 3

Basic orbit enumeration

This package contains a new implementation of the standard orbit enumeration algorithm. The design principles for this implementation have been:

- Allow partial orbit enumeration and later continuation.
- Consequently use hashing techniques.
- Implement stabiliser calculation and Schreier transversals on demand.
- Allow for searching in orbits during orbit enumeration.

Some of these design principles made it necessary to change the user interface in comparison to the standard GAP one.

3.1 Enumerating orbits

The enumeration of an orbit works in at least two stages: First an orbit object is created with all the necessary information to describe the orbit. Then the actual enumeration is started. The latter stage can be repeated as many times as needed in the case that the orbit enumeration stopped for some reason before the orbit was enumerated completely. See below for conditions under which this happens.

For orbit object creation there is the following function:

3.1.1 Orb

◇ `Orb(`*`gens, point, op[, opt]`*`)` (function)

Returns: An orbit object

The argument *`gens`* is either a GAP group object or a list of generators of the group acting, *`point`* is a point in the orbit to be enumerated, *`op`* is a GAP function describing the action of the generators on points in the usual way, that is, *`op`*`(p,g)` returns the result of the action of the element `g` on the point `p`.

The optional argument *`opt`* is a GAP record which can contain quite a few options changing the orbit enumeration. For a list of possible options see Subsection 3.1.4 at the end of this section.

The function returns an “orbit” object that can later be used to enumerate (a part of) the orbit of *`point`* under the action of the group generated by *`gens`*.

If *gens* is a group object, then its generators are taken as the list of generators acting. If the group object knows its size, then this size is used to speed up orbit and in particular stabiliser computations.

The following operation actually starts the orbit enumeration:

3.1.2 Enumerate

♢ `Enumerate(`*orb*`[,` *limit*`])` (operation)

Returns: The orbit object *orb*

orb must be an orbit object created by `Orb` (3.1.1). The optional argument *limit* must be a positive integer meaning that the orbit enumeration should stop if *limit* points have been found, regardless whether the orbit is complete or not. Note that the orbit enumeration can be continued by again calling the `Enumerate` operation. If the argument *limit* is omitted, the whole orbit is enumerated, unless other options lead to prior termination.

To see whether an orbit is closed you can use the following filter:

3.1.3 IsClosed

♢ `IsClosed(`*orb*`)` (filter)

Returns: `true` or `false`

The result indicates, whether the orbit *orb* is already complete or not.

Here is an example of an orbit enumeration:

Example

```
gap> g := GeneratorsOfGroup(MathieuGroup(24));
[ (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23),
  (3,17,10,7,9)(4,13,14,19,5)(8,18,11,12,23)(15,20,22,21,16),
  (1,24)(2,23)(3,12)(4,16)(5,18)(6,10)(7,20)(8,14)(9,21)(11,17)
  (13,22)(15,19)
 ]
gap> o := Orb(g,2,OnPoints);
<open Int-orbit, 1 points>
gap> Enumerate(o,20);
<open Int-orbit, 21 points>
gap> IsClosed(o);
false
gap> Enumerate(o);
<closed Int-orbit, 24 points>
gap> IsClosed(o);
true
```

The orbit object `o` now behaves like an immutable dense list, the entries of which are the points in the orbit in the order as they were found during the orbit enumeration (note that this is not always true when one uses the function `AddGeneratorsToOrbit` (3.1.15)). So you can ask the orbit for its length, access entries, and ask, whether a given point lies in the orbit or not. Due to the hashing techniques used such lookups are quite fast, they usually only use a constant time regardless of the length of the orbit!

Example

```
gap> Length(o);
24
gap> o[1];
2
```

```
gap> o[2];
3
gap> o{[3..5]};
[ 23, 4, 17 ]
gap> 17 in o;
true
gap> Position(o,17);
5
```

3.1.4 Options for orbits

The optional fourth argument *opt* of the function `Orb` (3.1.1) is a GAP record and its components change the behaviour of the orbit enumeration. In this subsection we explain the use of the components of this options record. All components are themselves optional. For every component we also describe the possible values in the following list:

`eqfunc` This component always has to be given together with the component `hashfunc`. If both are given, they are used to set up a hash table to store the points in the orbit. You have to use this if the automatic mechanism to find a suitable hash function does not work for your starting point in the orbit.

Note that if you use this feature, the hash table cannot grow automatically any more, unless you also use the components `hfbig` and `hfdbig` as well. See the description of `GrowHT` (4.4.5) for an explanation how to use this feature.

`genstoapply` This is only used internally and is intentionally not documented.

`grpsizebound` Possible values for this component are positive integers. By setting this value one can help the orbit enumeration to complete earlier. The given number must be an upper bound for the order of the group. If the exact group order is given and the stabiliser is calculated during the orbit enumeration (see component `permgens`), then the orbit enumeration can stop as soon as the orbit is found completely and the stabiliser is complete, which is usually much earlier than after all generator are applied to all points in the orbit.

`hashfunc` See component `eqfunc`.

`hashlen` Possible values are positive integers. This component determines the initial size of the hash used for the orbit enumeration. The default value is 10000. If the hash table turns out not to be large enough, it is automatically increased by a factor of two during the calculation. Although this process is quite fast it still improves performance to give a sensible hash size in advance.

`hfbig` and `hfdbig` These components can only be used in connection with `eqfunc` and `hashfunc` and are otherwise ignored. There values are simply passed on to the hash table created. The idea is to still be able to grow the hash table if need be. See Section 4.5 for more details.

`treehashsize` This component indicates that instead of a normal hash table a tree hash table (TreeHashTab) should be used (see Section 4.1). If bound, it must be set to the length of the tree hash table. You should still choose this length big enough, however, this type of hash table should be more resilient to bad hash functions since the performance of operations will only deteriorate up to $log(n)$ instead of to n (number of entries). If you use this option your hash keys must be comparable by $<$ and not only by `=`. You can supply your own three-way comparison function (see `HTCreate` (4.3.1)) by using the `cmpfunc` component.

`cmpfunc` If the previous component `treehashsize` is bound, you can specify a three-way comparison function for the hash keys in this component. See `HTCreate` (4.3.1) and `AVLCmp` (8.2.2) for details.

`log` If this component is set to `true` then a log of the enumeration of the orbit is written into the components `log`, `logind` and `logpos`. Every time a new point is found in the orbit enumeration, two numbers are appended to the log, first the number of the generator applied, then the index, under which the new point is stored in the orbit. For each point in the orbit, the start of the entries for that point in `log` is stored in `logind` and the end of those entries is marked by storing the number of the last generator producing a new point negated.

The purpose of a log is the following: With a log one can later add group generators to the orbit and thus get a different Schreier tree, such that the resulting orbit enumeration is still a breadth first enumeration using the new generating set! This is desirable to decrease the depth of the Schreier tree. The log helps to implement this in a way, such that the old generators do not again have to be applied to all the points in the orbit. See `AddGeneratorsToOrbit` (3.1.15) for details.

A log needs roughly 3 machine words per point in the orbit as memory.

`lookingfor` This component is used to search for something in the orbit. The idea is that the orbit enumeration is stopped when some condition is met. This condition can be specified with a great flexibility. The first way is to store a list of points into `orb.lookingfor`. In that case the orbit enumeration stops, when a point is found that is in that list. A second possiblity is to store a hash table object into `orb.lookingfor`. Then every newly found point in the orbit is looked up in that hash table and the orbit enumeration stops as soon as a point is found that is also in the hash table. The third possibility is functional: You can store a GAP function into `opt.lookingfor` which is called for every newly found point in the orbit. It gets both the orbit object and the point as its two arguments. This function has to return `false` or `true` and in the latter case the orbit enumeration is stopped.

Whenever the orbit enumeration is stopped the component `found` is set to the number of the found point in the orbit. Access this information using `PositionOfFound(orb)`.

`matgens` This is not yet implemented. It will allow for stabiliser computations in matrix groups.

`onlystab` If this boolean flag is set to `true` then the orbit enumeration stops once the stabiliser is completely determined. Note that this can only be known, if a bound for the group size is given in the `opt.grpsizebound` option and when more than half of the orbit is already found, or when `opt.stabsizebound` is given.

`orbsizebound` Possible values for this component are positive integers. The given number must be an upper bound for the orbit length. Giving this number helps the orbit enumeration to stop earlier, when the orbit is found completely.

`permbase` This component is used to tell the orbit enumerator that a certain list of points is a base of the permutation representation given in the `opt.permgens` component. This information is often available beforehand and can drastically speed up the calculation of Schreier generators, especially for the common case that they are trivial. The value is just a list of integers.

permgens If this component is set, it must be set to a list of permutations, that represent the same group as the generators used to define the orbit. This permutation representation is then used to calculate the stabiliser of the starting point. After the orbit enumeration is complete, you can call `Stabilizer(`*orb*`)` with *orb* being the orbit object and get the stabiliser as a permutation group. The stabiliser is also stored in the `stab` component of the orbit object. Furthermore, the size of the stabiliser is stored in the `stabsize` component of the orbit object and the component `stabwords` contains the stabiliser generators as words in the original group generators. Access these words with `StabWords(orb)`. Here, a word is a list of integers, where positive integers are numbers of generators and a negative integer i indicates the inverse of the generator with number $-i$. In this way, complete information about the stabiliser can be derived from the orbit object.

report Possible values are non-negative integers. This value asks for a status report whenever the orbit enumeration has applied all generators to `opt.report` points. A value of 0, which is the default, switches off this report. In each report, the total number of points already found are given.

schreier This boolean flag decides, whether a Schreier tree is stored together with the orbit. A Schreier tree just stores for each point, which generator was applied to which other point in the orbit to get it. Thus, having the Schreier tree enables the usage of the operations `TraceSchreierTreeForward` (3.1.11) and `TraceSchreierTreeBack` (3.1.12). A Schreier tree needs two additional machine words of memory per point in the orbit. The `opt.schreier` flag is automatically set when a stabiliser is computed during orbit enumeration (see components `opt.permgens` and `opt.matgens`).

schreiergenaction The value of this component must be a function with 4 arguments: the orbit object, an index *i*, an integer *j*, and an index *pos*. It is called, whenever during the orbit enumeration generator number *j* was applied to point number *i* and the result was an already known point with number *pos*. This is no longer done, once the component `stabcomplete` is set to `true`, which happens when there is evidence that the stabiliser is already completely determined.

This component is used internally when the `permgens` component was set and the stabiliser is calculated.

stab This component is used to tell the orbit enumerator that a subgroup of the stabiliser of the starting point is already known. Store a subgroup of the group generated by the permutations in `opt.permgens` stabilising the starting point into this component.

stabchainrandom This value can be a positive integer between 1 and 1000. If `opt.permgens` is given, an integer value is used to set the `random` option when calculating a stabiliser chain to compute the size of the group generated by the Schreier generators. Although this size computation can be speeded up considerably, the user should be aware that for values smaller than 1000 this triggers a Monte Carlo algorithm that can produce wrong results with a certain error probability. A verification of the obtained results is advisable. Note however, that such computations can only err in one direction, namely underestimating the size of the group.

stabsizebound Possible values for this component are positive integers. The given number must be an upper bound for the size of the stabiliser. Giving this number helps the orbit enumeration to stop earlier, when also `opt.orbsizebound` or `opt.grpsizebound` are given or when `opt.onlystab` is set.

storenumbers This boolean flag decides, whether the positions of points in the orbit are stored in the hash. The memory requirement for this is one machine word (4 or 8 bytes depending on the architecture) per point in the orbit. If you just need the orbit itself this is not necessary. If you however want to find the position of a point in the orbit efficiently after enumeration, then you should switch this on. That is, the operation `\in` is always fast, but `Position(`*`orb`*`,` *`point`*`)` is only fast if `opt.storenumbers` was set to `true` or the orbit is “permutations acting on positive integers”. In the latter case this flag is ignored.

For some examples using these options see Chapter 11.

3.1.5 Output components of orbits

The following components are bound in an orbit object. There might be some more, but those are implementation specific and not guaranteed to be there in future versions. Note that you have to access these components using the “.˜” dot exclamation mark notation and you should avoid using these if at all possible.

depth and depthmarks If the orbit has either a Schreier tree or a log, then the component `depth` holds its depth, that is the maximal number of generator applications needed to reach any point in the orbit. The corresponding component `depthmarks` is a list of indices, at position i it holds the index of the first point in the orbit in depth i in the Schreier tree.

gens The list of group generators.

ht If the orbit uses a hash table it is stored in this component.

op The operation function.

orbind If generators have been added to the orbit later then the order in which the points are actually stored in the orbit might not correspond to a breadth first search. To cover this case, the component `orbind` contains in position i the index under which the i-th point in the breadth-first search using the new generating set is actually stored in the orbit.

schreiergen and schreierpos If a Schreier tree of the orbit was kept then both these components are lists containing integers. If point number i was found by applying generator number j to point number p then position i of `schreiergen` is j and position i of `schreierpos` is p. You can use the operations `TraceSchreierTreeForward` (3.1.11) and `TraceSchreierTreeBack` (3.1.12) to compute words in the generators using these two components.

tab For an orbit in which permutations act on positive integers this component is bound to a list containing in position i the index in the orbit, where the number i is stored.

The following operations help to ask additional information about orbit objects:

3.1.6 StabWords (basic)

◊ `StabWords(`*orb*`)` (operation)

Returns: A list of words

If the stabiliser was computed during the orbit enumeration, then this function returns the stabiliser generators found as words in the generators. A word is a sequence of integers, where positive integers stand for generators and negative numbers for their inverses.

3.1.7 PositionOfFound

◊ `PositionOfFound(`*orb*`)` (operation)

Returns: An integer

If during the orbit enumeration the option `lookingfor` was used and the orbit enumerator looked for something, then this operation returns the index in the orbit, where the something was found most recently.

3.1.8 DepthOfSchreierTree

◊ `DepthOfSchreierTree(`*orb*`)` (operation)

Returns: An integer

If a Schreier tree or a log was stored during orbit enumeration, then this operation returns the depth of the Schreier tree.

We present a few more operations one can do with orbit objects. One can express the action of a given group element in the group generated by the generators given in the `Orb` command on this orbit as a permutation:

3.1.9 ActionOnOrbit

◊ `ActionOnOrbit(`*orb, grpels*`)` (operation)

Returns: A permutation or `fail`

orb must be an orbit object and *grpels* a list of group elements acting on the orbit. This operation calculates the action of *grpels* on *orb* as GAP permutations, where the numbering of the points is in the same order as the points have been found in the orbit. Note that this operation is particularly fast if the orbit is an orbit of a permutation group acting on positive integers or if you used the option `storenumbers` described in Subsection 3.1.4.

3.1.10 OrbActionHomomorphism

◊ `OrbActionHomomorphism(`*g, orb*`)` (operation)

Returns: An action homomorphism

The argument *g* must be a group and *orb* an orbit on which *g* acts in the action of the orbit object. This operation returns a homomorphism into a permutation group acquired by taking the action of *g* on the orbit.

3.1.11 TraceSchreierTreeForward

◊ `TraceSchreierTreeForward(`*orb, nr*`)` (operation)

Returns: A word in the generators

`orb` must be an orbit object with a Schreier tree, that is, the option `schreier` must have been set during creation, and `nr` must be the number of a point in the orbit. This operation traces the Schreier tree and returns a word in the generators that maps the starting point to the point with number `nr`. Here, a word is a list of integers, where positive integers are numbers of generators and a negative integer i indicates the inverse of the generator with number $-i$.

3.1.12 TraceSchreierTreeBack

◊ `TraceSchreierTreeBack(orb, nr)` (operation)

Returns: A word in the generators

`orb` must be an orbit object with a Schreier tree, that is, the option `schreier` must have been set during creation, and `nr` must be the number of a point in the orbit. This operation traces the Schreier tree and returns a word in the generators that maps the point with number `nr` to the starting point. As above, a word is here a list of integers, where positive integers are numbers of generators and a negative integer i indicates the inverse of the generator with number $-i$.

3.1.13 ActWithWord

◊ `ActWithWord(gens, w, op, p)` (operation)

Returns: A point

`gens` must be a list of group generators, `w` a list of positive integers less than or equal to the length of `gens`, `op` an action function and `p` a point. This operation computes the action of the word `w` in the generators `gens` on the point `p` and returns the result.

3.1.14 EvaluateWord

◊ `EvaluateWord(gens, w)` (operation)

Returns: A group element

`gens` must be a list of group generators, `w` a list of positive integers less than or equal to the length of `gens`. This operation evaluates the word `w` in the generators `gens` and returns the result.

3.1.15 AddGeneratorsToOrbit

◊ `AddGeneratorsToOrbit(orb, l[, p])` (operation)

Returns: The orbit object `orb`

`orb` must be an orbit object, `l` a list of new generators and, if given, `p` must be a list of permutations of equal length. `p` must be given if and only if the component `permgens` was specified upon creation of the orbit object. The new generators are appended to the old list of generators. The orbit object is changed such that it then shows the outcome of a breadth-first orbit enumeration with the *new* list of generators. Note that the order of the points already enumerated will *not* be changed. However, the Schreier tree changes, the component `orbind` is changed to indicate the order in which the points were found in the breadth-first search with the new generators and the components `depth` and `depthmarks` are changed.

Note that all this is particularly efficient if the orbit has a log. If you add generators to an orbit with log, the old generators do not have to be applied again to all points!

Note that new generators can actually enlarge an orbit if they generate a larger group than the old ones alone. Note also that when adding generators, the orbit is automatically enumerated completely

3.1.16 MakeSchreierTreeShallow

◇ `MakeSchreierTreeShallow(`*`orb[, d]`*`)` (operation)

Returns: The orbit object *orb*

Uses `AddGeneratorsToOrbit` (3.1.15) to add more generators to the orbit in order to make the Schreier tree shallower. If *d* it is given, generators are added until the depth is less than or equal to *d* or until three more generators did not reduce the depth any more. If *d* is not given, then the logarithm to base 2 of the orbit length is taken as a default value.

3.1.17 FindSuborbits

◇ `FindSuborbits(`*`orb, subgens[, nrsuborbits]`*`)` (operation)

Returns: A record

The argument *orb* must be a closed orbit object with a Schreier vector, *subgens* a list of generators for a subgroup of the originally acting group. If given, *nrsuborbits* must be a lower limit for the number of suborbits.

The returned record describes the suborbit structure of *orb* with respect to the group generated by *subgens* using the following components: `issuborbitrecord` is bound to `true`, `o` is bound to *orb*, `nrsuborbits` is bound to the number of suborbits and `reps` is a list of length `nrsuborbits` containing the index in the orbit of a representative for each suborbit. Likewise, `words` contains words in the original group generators of the orbit that map the starting point of the orbit to those representatives. `lens` is a list containing the lengths of the suborbits. The component `suborbs` is bound to a list of lists, one for each suborbit containing the indices of the points in the orbit. The component `suborbnr` is a list with the same length as the orbit, containing in position *i* the number of the suborbit in which point *i* in the orbit is contained.

Finally, the component `conjsuborbit` is bound to a list of length `nrsuborbits`, containing for each suborbit the number the suborbit reached from the starting point by the inverse of the word used to reach the orbit representative. This latter information probably only makes sense when the subgroup generated by *subgens* is contained in the point stabiliser of the starting point of the orbit, because then this is the so-called conjugate suborbit of a suborbit.

3.1.18 OrbitIntersectionMatrix

◇ `OrbitIntersectionMatrix(`*`r, g`*`)` (operation)

Returns: An integer matrix

The argument *r* must be a suborbit record as returned by the operation `FindSuborbits` (3.1.17) above, describing the suborbit structure of an orbit with respect to a subgroup. *g* must be an element of the acting group. If k is the number of suborbits and the suborbits are $O_1,\ldots,O_k$, then the matrix returned by this operation has the integer $|O_i \cdot g \cap O_j|$ in its (i,j)-entry.

3.1.19 ORB_EstimateOrbitSize

◇ `ORB_EstimateOrbitSize(`*`gens, pt, op, L, limit, timeout`*`)` (function)

Returns: `fail` or a record

The argument *gens* is a list of group generators for a group G, the argument *pt* a point and *op* and action function for a group action of G acting on points like *pt*. This function starts to act with random elements of G on *pt* producing random elements of the orbit $pt * G$ and uses the birthday paradox to estimate the orbit size. To this end it creates points of the orbit until *L* coincidences (points found twice) have been found. If before this happens `*limit*` tries have been reached or if more than `*timeout*` milliseconds have ellapsed, the function gives up and returns `fail`. Otherwise it estimates the orbit size giving an estimate in the component `estimate`, a confidence interval described by the components `lowerbound` and `upperbound`, a list of generators for the stabiliser in the component `Sgens` and the number of coincidences that were caused by picking the same group element. The length of `Sgens` is *L*−`grpcoinc`. Use at least 15 for *L*, otherwise the statistics are not valid.

Chapter 4

Hashing techniques

4.1 The idea of hashing

If one wants to store a certain set of similar objects and wants to quickly access a given one (or come back with the result that it is unknown), the first idea would be to store them in a list, possibly sorted for faster access. This however still would need $\log(n)$ comparisons to find a given element or to decide that it is not yet stored.

Therefore one uses a much bigger array and uses a function on the space of possible objects with integer values to decide, where in the array to store a certain object. If this so called hash function distributes the actually stored objects well enough over the array, the access time is constant in average. Of course, a hash function will usually not be injective, so one needs a strategy what to do in case of a so-called "collision", that is, if more than one object with the same hash value has to be stored. This package provides two ways to deal with collisions, one is implemented in the so called "HashTabs" and another in the "TreeHashTabs". The former simply uses other parts of the array to store the data involved in the collisions and the latter uses an AVL tree (see Chapter 8) to store all data objects with the same hash value. Both are used basically in the same way but sometimes behave a bit differently.

The basic functions to work with hash tables are `HTCreate` (4.3.1), `HTAdd` (4.3.2), `HTValue` (4.3.3), `HTDelete` (4.3.5) and `HTUpdate` (4.3.4). They are described in Section 4.3.

The legacy functions from older versions of this package to work with hash tables are `NewHT` (4.4.1), `AddHT` (4.4.2), and `ValueHT` (4.4.3). They are described in Section 4.4. In the next section, we first describe the infrastructure for hash functions.

4.2 Hash functions

In the orb package hash functions are chosen automatically by giving a sample object together with the length of the hash table. This is done with the following operation:

4.2.1 ChooseHashFunction

◊ `ChooseHashFunction(`*ob*`, `*len*`)` (operation)

Returns: a record

The first argument *ob* must be a sample object, that is, an object like those we want to store in the hash table later on. The argument *len* is an integer that gives the length of the hash table. Note that this might be called later on automatically, when a hash table is increased in size. The operation returns a record with two components. The component `func` is a GAP function taking two arguments, see below. The component `data` is some GAP object. Later on, the hash function will be called with two arguments, the first is the object for which it should call the hash value and the second argument must be the data stored in the `data` component.

The hash function has to return values between 1 and the hash length *`len`* inclusively.

This setup is chosen such that the hash functions can be global objects that are not created during the execution of `ChooseHashFunction` but still can change their behaviour depending on the data.

In the following we just document, for which types of objects there are hash functions that can be found using `ChooseHashFunction` (4.2.1).

4.2.2 ChooseHashFunction (gf2vec)

◊ `ChooseHashFunction(`*`ob, len`*`)` (method)

Returns: a record

This method is for compressed vectors over the field `GF(2)` of two elements. Note that there is no hash function for non-compressed vectors over `GF(2)` because those objects cannot efficiently be recognised from their type.

Note that you can only use the resulting hash functions for vectors of the same length.

4.2.3 ChooseHashFunction (8bitvec)

◊ `ChooseHashFunction(`*`ob, len`*`)` (method)

Returns: a record

This method is for compressed vectors over a finite field with up to 256 elements. Note that there is no hash function for non-compressed such vectors because those objects cannot efficiently be recognised from their type.

Note that you can only use the resulting hash functions for vectors of the same length.

4.2.4 ChooseHashFunction (gf2mat)

◊ `ChooseHashFunction(`*`ob, len`*`)` (method)

Returns: a record

This method is for compressed matrices over the field `GF(2)` of two elements. Note that there is no hash function for non-compressed matrices over `GF(2)` because those objects cannot efficiently be recognised from their type.

Note that you can only use the resulting hash functions for matrices of the same size.

4.2.5 ChooseHashFunction (8bitmat)

◊ `ChooseHashFunction(`*`ob, len`*`)` (method)

Returns: a record

This method is for compressed matrices over a finite field with up to 256 elements. Note that there is no hash function for non-compressed such vectors because those objects cannot efficiently be recognised from their type.

Note that you can only use the resulting hash functions for matrices of the same size.

4.2.6 ChooseHashFunction (int)

◊ `ChooseHashFunction(`*`ob, len`*`)` (method)

Returns: a record

This method is for integers.

4.2.7 ChooseHashFunction (perm)

◊ `ChooseHashFunction(`*`ob, len`*`)` (method)

Returns: a record

This method is for permutations.

4.2.8 ChooseHashFunction (intlist)

◊ `ChooseHashFunction(`*`ob, len`*`)` (method)

Returns: a record

This method is for lists of integers.

4.2.9 ChooseHashFunction (NBitsPcWord)

◊ `ChooseHashFunction(`*`ob, len`*`)` (method)

Returns: a record

This method is for kernel Pc words.

4.2.10 ChooseHashFunction (IntLists)

◊ `ChooseHashFunction(`*`ob, len`*`)` (method)

Returns: a record

This method is for lists of integers.

4.2.11 ChooseHashFunction (MatLists)

◊ `ChooseHashFunction(`*`ob, len`*`)` (method)

Returns: a record

This method is for lists of matrices.

4.3 Using hash tables

4.3.1 HTCreate

◊ `HTCreate(`*`sample[, opt]`*`)` (operation)

Returns: a new hash table object

A new hash table for objects like *`sample`* is created. The second argument *`opt`* is an optional options record, which will supplied in most cases, if only to specify the length and type of the hash table to be used. The following components in this record can be bound:

`treehashsize` If this component is bound the type of the hash table is a TreeHashTab. The value must be a positive integer and will be the size of the hash table. Note that for this type of hash table the keys to be stored in the hash must be comparable using $<$. A three-way comparison function can be supplied using the component `cmpfunc` (see below).

`treehashtab` If this component is bound the type of the hash table is a TreeHashTab. This option is superfluous if `treehashsize` is used.

`hf` and `hfd` If these components are bound, they are used as the hash function. The value of `hf` must be a function taking two arguments, the first being the object for which the hash function shall be computed and the second being the value of `hfd`. The returned value must be an integer in the range from 1 to the length of the hash. If either of these components is not bound, an automatic choice for the hash function is done using `ChooseHashFunction` (4.2.1) and the supplied sample object *`sample`*.

Note that if you specify these two components and are using a HashTab table then this table cannot grow unless you also bind the components `hfbig`, `hfdbig` and `cangrow`.

`cmpfunc` This component can be bound to a three-way comparison function taking two arguments *`a`* and *`b`* (which will be keys for the TreeHashTab) and returns -1 if $a < b$, 0 if $a = b$ and 1 if $a > b$. If this component is not bound the function `AVLCmp` (8.2.2) is taken, which simply calls the generic operations $<$ and = to do the job.

`hashlen` If this component is bound the type of the hash table is a standard HashTab table. That is, collisions are dealt with by storing additional entries in other slots. This is the traditional way to implement a hash table. Note that currently deleting entries in such a hash table is not implemented, since it could only be done by leaving a “deleted” mark which could pollute that hash table. Use TreeHashTabs instead if you need deletion. The value bound to `hashlen` must be a positive integer and will be the initial length of the hash table.

Note that it is a good idea to choose a prime number as the hash length due to the algorithm for collision handling which works particularly well in that case. The hash function is chosen automatically.

`hashtab` If this component is bound the type of the hash table is a standard HashTab table. This component is superfluous if `hashlen` is bound.

`eqf` For HashTab tables the function taking two arguments bound to this component is used to compare keys in the hash table. If this component is not bound the usual = operation is taken.

`hfbig` and `hfdbig` and `cangrow` If you have used the components `hf` and `hfd` then your hash table cannot automatically grow when it fills up. This is because the length of the table is built into the hash function. If you still want your hash table to be able to grow automatically, then bind a hash function returning arbitrary integers to `hfbig`, the corresponding data for the second argument to `hfdbig` and bind `cangrow` to `true`. Then the hash table will automatically grow and take this new hash function modulo the new length of the hash table as hash function.

4.3.2 HTAdd

◊ `HTAdd(`*`ht, key, value`*`)` (operation)

Returns: a hash value

Stores the object *`key`* into the hash table *`ht`* and stores the value *`val`* together with *`ob`*. The result is `fail` if an error occurred, which can be that an object equal to *`key`* is already stored in the hash table or that the hash table is already full. The latter can only happen, if the hash table is no TreeHashTab and cannot grow automatically.

If no error occurs, the result is an integer indicating the place in the hash table where the object is stored. Note that once the hash table grows automatically this number is no longer the same!

If the value *val* is `true` for all objects in the hash, no extra memory is used for the values. All other values are stored in the hash. The value `fail` cannot be stored as it indicates that the object is not found in the hash.

See Section 4.5 for details on the data structures and especially about memory requirements.

4.3.3 HTValue

◇ `HTValue(`*ht*`,` *key*`)` (operation)

Returns: `fail` or `true` or a value

Looks up the object *key* in the hash table *ht*. If the object is not found, `fail` is returned. Otherwise, the value stored with the object is returned. Note that if this value was `true` no extra memory is used for this.

4.3.4 HTUpdate

◇ `HTUpdate(`*ht*`,` *key*`,` *value*`)` (operation)

Returns: `fail` or `true` or a value

The object *key* must already be stored in the hash table *ht*, otherwise this operation returns `fail`. The value stored with *key* in the hash is replaced by *value* and the previously stored value is returned.

4.3.5 HTDelete

◇ `HTDelete(`*ht*`,` *key*`)` (operation)

Returns: `fail` or `true` or a value

The object *key* along with its stored value is removed from the hash table *ht*. Note that this currently only works for TreeHashTabs and not for HashTab tables. It is an error if *key* is not found in the hash table and `fail` is returned in this case.

4.3.6 HTGrow

◇ `HTGrow(`*ht*`,` *ob*`)` (function)

Returns: nothing

This is a more or less internal operation. It is called when the space in a hash table becomes scarce. The first argument *ht* must be a hash table object, the second a sample point. The function increases the hash size by a factor of 2. This makes it necessary to choose a new hash function. Usually this is done with the usual `ChooseHashFunction` method. However, one can bind the two components `hfbig` and `hfdbig` in the options record of `HTCreate` (4.3.1) to a function and a record respectively and bind `cangrow` to `true`. In that case, upon growing the hash, a new hash function is created by taking the function `hfbig` together with `hfdbig` as second data argument and reducing the resulting integer modulo the hash length. In this way one can specify a hash function suitable for all hash sizes by simply producing big enough hash values.

4.4 Using hash tables (legacy code)

Note that the functions described in this section are obsolete since version 3.0 of orb and are only kept for backward compatibility. Please use the functions in Section 4.3 in new code.

The following functions are needed to use hash tables. For details about the data structures see Section 4.5.

4.4.1 NewHT

◊ `NewHT(`*`sample, len`*`)` (function)

Returns: a new hash table object

A new hash table for objects like *`sample`* of length *`len`* is created. Note that it is a good idea to choose a prime number as the hash length due to the algorithm for collision handling which works particularly well in that case. The hash function is chosen automatically. The resulting object can be used with the functions `AddHT` (4.4.2) and `ValueHT` (4.4.3). It will start with length *`len`* but will grow as necessary.

4.4.2 AddHT

◊ `AddHT(`*`ht, ob, val`*`)` (function)

Returns: an integer or fail

Stores the object *`ob`* into the hash table *`ht`* and stores the value *`val`* together with *`ob`*. The result is `fail` if an error occurred, which can only be that the hash table is already full. This can only happen, if the hash table cannot grow automatically.

If no error occurs, the result is an integer indicating the place in the hash table where the object is stored. Note that once the hash table grows automatically this number is no longer the same!

If the value *`val`* is `true` for all objects in the hash, no extra memory is used for the values. All other values are stored in the hash. The value `fail` cannot be stored as it indicates that the object is not found in the hash.

See Section 4.5 for details on the data structures and especially about memory requirements.

4.4.3 ValueHT

◊ `ValueHT(`*`ht, ob`*`)` (function)

Returns: the stored value, `true`, or `fail`

Looks up the object *`ob`* in the hash table *`ht`*. If the object is not found, `fail` is returned. Otherwise, the value stored with the object is returned. Note that if this value was `true` no extra memory is used for this.

The following function is only documented for the sake of completeness and for emergency situations, where `NewHT` (4.4.1) tries to be too intelligent.

4.4.4 InitHT

◊ `InitHT(`*`len, hfun, eqfun`*`)` (function)

Returns: a new hash table object

This is usually only an internal function. It is called from `NewHT` (4.4.1). The argument *`len`* is the length of the hash table, *`hfun`* is the hash function record as returned by `ChooseHashFunction` (4.2.1) and *`eqfun`* is a comparison function taking two arguments and returning `true` or `false`.

Note that automatic growing is switched on for the new hash table which means that if the hash table grows, a new hash function is chosen using `ChooseHashFunction` (4.2.1). If you do not want this, change the component `cangrow` to `false` after creating the hash table.

4.4.5 GrowHT

◊ `GrowHT(`*ht*`,` *ob*`)` (function)

Returns: nothing

This is a more or less internal function. It is called when the space in a hash table becomes scarce. The first argument *ht* must be a hash table object, the second a sample point. The function increases the hash size by a factor of 2 for hash tables and 20 for tree hash tables. This makes it necessary to choose a new hash function. Usually this is done with the usual `ChooseHashFunction` method. However, one can assign the two components `hfbig` and `hfdbig` to a function and a record respectively. In that case, upon growing the hash, a new hash function is created by taking the function `hfbig` together with `hfdbig` as second data argument and reducing the resulting integer modulo the hash length. In this way one can specify a hash function suitable for all hash sizes by simply producing big enough hash values.

4.5 The data structures for hash tables

A legacy hash table object is just a record with the following components:

`els` A GAP list storing the elements. Its length can be as long as the component `len` indicates but will only grow as necessary when elements are stored in the hash.

`vals` A GAP list storing the corresponding values. If a value is `true` nothing is stored here to save memory.

`len` Length of the hash table.

`nr` Number of elements stored in the hash table.

`hf` The hash function (value of the `func` component in the record returned by `ChooseHashFunction` (4.2.1)).

`hfd` The data for the second argument of the hash function (value of the `data` component in the record returned by `ChooseHashFunction` (4.2.1)).

`eqf` A comparison function taking two arguments and returning `true` for equality or `false` otherwise.

`collisions` Number of collisions (see below).

`accesses` Number of lookup or store accesses to the hash.

`cangrow` A boolean value indicating whether the hash can grow automatically or not.

`ishash` Is `true` to indicate that this is a hash table record.

`hfbig` and `hfdbig` Used for hash tables which need to be able to grow but where the user supplied the hash function. See Section `HTCreate` (4.3.1) for more details.

A new style HashTab objects are component objects with the same components except that there is no component `ishash` since these objects are recognised by their type.

A TreeHashTab is very similar. It is a positional object with basically the same components, except that `eqf` is replaced by the three-way comparison function `cmpfunc`. Since TreeHashTabs do not grow, the components `hfbig`, `hfdbig` and `cangrow` are never bound. Each slot in the `els` component is either unbound (empty), or bound to the only key stored in the hash which has this hash value or, if there is more than one key for that hash value, the slot is bound to an AVL tree containing all such keys (and values).

4.5.1 Memory requirements

Due to the data structure defined above the hash table will need one machine word (4 bytes on 32bit machines and 8 bytes on 64bit machines) per possible entry in the hash if all values corresponding to objects in the hash are `true` and two machine words otherwise. This means that the memory requirement for the hash itself is proportional to the hash table length and not to the number of objects actually stored in the hash!

In addition one of course needs the memory to store the objects themselves.

For TreeHashTabs there are additional memory requirements. As soon as there are more than one key hashing to the same value, the memory for an AVL tree object is needed in addition. An AVL tree objects needs about 10 machine words for the tree object and then another 4 machine words for each entry stored in the tree. Note that for many collisions this can be significantly more than for HashTab tables. However, the advantage of TreeHashTabs is that even for a bad hash function the performance is never worse than $log(n)$ for each operation where n is the number of keys in the hash with the same hash value.

4.5.2 Handling of collisions

This section is only relevant for HashTab objects.

If two or more objects have the same hash value, the following is done: If the hash value is coprime to the hash length, the hash value is taken as “the increment”, otherwise 1 is taken. The code to find the proper place for an object just repeatedly adds the increment to the current position modulo the hash length. Due to the choice of the increment this will eventually try all places in the hash table. Every such increment step is counted as a collision in the `collisions` component in the hash table. This algorithm explains why it is sensible to choose a prime number as the length of a hash table.

4.5.3 Efficiency

Hashing is efficient as long as there are not too many collisions. It is not a problem if the number of collisions (counted in the `collisions` component) is smaller than the number of accesses (counted in the `accesses` component).

A high number of collisions can be caused by a bad hash function, because the hash table is too small (do not fill a hash table to more than about 80%), or because the objects to store are just not well enough distributed. Hash tables will grow automatically if too many collisions are detected or if they are filled to 80%.

The advantage of TreeHashTabs is that even for a bad hash function the performance is never worse than $log(n)$ for each operation where n is the number of keys in the hash with the same hash value. However, they need a bit more memory.

Chapter 5

Caching techniques

5.1 The idea of caching

If one wants to work with a large number of large objects which require some time to prepare and one does not know beforehand, how often one will need each one, it makes sense to work with some sort of cache. A cache is a data structure to keep some of the objects already produced but not too many of them to waste a lot of memory. That is, objects which have not been used for some time can automatically be removed from the cache, whereas the objects which are used more frequently stay in the cache. This chapter describes an implementation of this idea used in the orbit-by-suborbit algorithms.

5.2 Using caches

A cache is created using the following operation:

5.2.1 LinkedListCache

◊ `LinkedListCache(`*`memorylimit`*`)` (operation)

Returns: A new cache object

This operation creates a new linked list cache that uses at most *`memorylimit`* bytes to store its entries. The cache is organised as a linked list, newly cached objects are appended at the beginning of the list, when the used memory grows over the limit, old objects are removed at the end of this list automatically.

New objects are entered into the hash with the following function:

5.2.2 CacheObject

◊ `CacheObject(`*`c, ob, mem`*`)` (operation)

Returns: A new node in the linked list cache

This operation enters the object *`ob`* into the cache *`c`*. The argument *`mem`* is an integer with the memory usage of the object *`ob`*. The object is prepended to the linked list cache and enough objects at the end are removed to enforce the memory usage limit.

5.2.3 ClearCache

◊ `ClearCache(c)` (operation)

Returns: Nothing

Completely clears the cache *c* removing all nodes.

A linked list cache is used as follows: Whenever you compute one of the objects you store it in the cache using `CacheObject` (5.2.2) and retain the linked list node that is returned. The usual place to retain it would be in a weak pointer object, such that this reference does not prevent the object to be garbage collected. When you next need this object, you check its corresponding position in the weak pointer object, if the reference is still there, you just use it and tell the cache that it was used again by calling `UseCacheObject` (5.2.4), otherwise you create it anew and store it in the cache again.

As long as the object stays in the cache it is not garbage collected and the weak pointer object will still have its reference. As soon as the object is thrown out of the cache, the only reference to its node is the weak pointer object, thus if a garbage collection happens, it can be garbage collected. Note that before that garbage collection happens, the object might still be accessible via the weak pointer object. In this way, the available main memory in the workspace is used very efficiently and can be freed immediately when needed.

5.2.4 UseCacheObject

◊ `UseCacheObject(c, r)` (operation)

Returns: Nothing

The argument *c* must be a cache object and *r* a node for such a cache. The object is either moved to the front of the linked list (if it is still in the cache) or it is re-cached. If necessary, objects at the end are removed from the cache to enforce the memory usage limit.

Chapter 6

Random elements

In this chapter we describe some fundamental mechanisms to produce (pseudo-) random elements that are used later in Chapter 7 about searching in groups and orbits.

6.1 Randomizing mutable objects

For certain types of mutable objects one can get a “random one” by calling the following operation:

6.1.1 Randomize

◊ `Randomize(ob[, rs])` (operation)

Returns: nothing

The mutable object *ob* is changed in place. The value afterwards is random. The optional second argument *rs* must be a random source and the random numbers used to randomize *ob* are created using the random source *rs* (see (**Reference: Random Sources**)). If *rs* is not given, then the global GAP random number generator is used.

Currently, there are `Randomize` methods for compressed vectors and compressed matrices over finite fields. See also the cvec package for methods for `cvec`s and `cmat`s.

For vectors and one-dimensional subspaces there are two special functions to create a list of random objects:

6.1.2 MakeRandomVectors

◊ `MakeRandomVectors(sample, number[, rs])` (function)

Returns: a list of random vectors

sample must be a vector for the mutable copies of which `Randomize` (6.1.1) is applicable and *number* must be a positive integer. If given, *rs* must be a random source. This function creates a list of *number* random vectors with the same type as *sample* using `Randomize` (6.1.1). For the creation of random numbers the random source *rs* is used or, if not given, the global GAP random number generator.

6.1.3 MakeRandomLines

◊ `MakeRandomLines(sample, number[, rs])` (function)

Returns: a list of normalised random vectors *sample* must be a vector for the mutable copies of which `Randomize` (6.1.1) is applicable and *number* must be a positive integer. If given, *rs* must be a random source. This function creates a list of *number* normalised random vectors with the same type as *sample* using `Randomize` (6.1.1). "Normalised" here means that the first non-zero entry in the vector is equal to 1. For the creation of random numbers the random source *rs* is used or, if not given, the global GAP random number generator.

6.2 Product replacement

For computations in finite groups product replacement algorithms are a viable method of generating pseudo-random elements. This section describes a framework and an object type to provide these algorithms. Roughly speaking a "product replacer object" is something that is created with a list of group generators and produces a sequence of pseudo random group elements using some random source for random numbers.

6.2.1 ProductReplacer

◊ `ProductReplacer(`*gens*`[,` *opt*`])` (operation)

Returns: a new product replacer object

gens must be a list of group generators. If given, *opt* is a GAP record with options. The operation creates a new product replacer object producing pseudo random elements in the group generated by the generators *gens*.

The exact algorithm used is explained below after the description of the options.

The following components in the options record have a defined meaning:

`randomsource` A random source object that is used to generate the random numbers used. If none is specified the global GAP random number generator is used.

`scramble` The `scramble` value in the algorithm described below can be set using this option. The default value is 30.

`scramblefactor` The `scramblefactor` value in the algorithm described below can be set using this option. The default value is 4.

`addslots` The `addslots` value in the algorithm described below can be set using this option. The default value is 5.

`maxdepth` If `maxdepth` is set, then the production of pseudo random elements starts all over whenever `maxdepth` product replacements have been performed. The rationale behind this is that the elements created should be evenly distributed but that the expressions in the generators should not be too long. A good compromise is usually to set `maxdepth` to 300 or 400.

`noaccu` Without this option set to `true` the "rattle" version of product replacement is used which involves an accumulator and uses two or three products per random element. To use the "shake" version with only one or two product replacement per random element set this component to `true`. The exact number of multiplications per random element also depends on the value of the `accelerator` component.

normalin There is a variant of the product replacement algorithm that produces elements in the normal closure of the group generated by a list of elements. It needs random elements in the ambient group in which the normal closure is defined. This is implemented here by setting the `normalin` component to a product replacer object working in the ambient group. In every step two elements a and b are picked and then a is either replaced by $a * b^c$ or $b^c * a$ (with equal probability), where c is a random element from the ambient group produced by the product replacer in the `normalin` component. It is recommended to switch off the accumulator and accelerator in the product replacer object for the ambient group. Then to produce one random element in the normal closure needs four multiplications.

accelerator If this option is set to `true` (which is the default), then the accelerator is used. This means that in each step two product replacement steps are performed, where both involve one distinguished slot called the "captain". The idea is that the current "team" of random elements uses one amongst them more often to increase the length of the words produced. See below for details of the algorithm with and without accelerator.

retirecaptain If this component is bound to a positive integer then the captain retires after so many steps of the algorithm. This is to use only two multiplications for each random element in the long run after proper mixing. The default value for `retirecaptain` is twice the scrambling time.

accus This component (default is 5) is the number of accumulators to use in the rattle variant. All accus are used in a round robin fashion. The purpose of multiple accus is to have a greater stochastical independence of adjacent random elements in the sequence.

The algorithm used does the following: A list of `Length(`*gens*`)+addslots` elements is created that starts with the elements *gens* and is filled up with random generators from *gens*. This element is called the "team". A product replacement without accelerator randomly chooses two elements in the list and replaces one of them by the product of the two. If an accelerator is used, then one product replacement step randomly chooses two slots i and j where $i, j > 1$ but $i = j$ is possible. Then first $l[1]$ is replaced by $l[1] * l[i]$ and after that $l[j]$ is replaced by $l[j] * l[1]$. The first team member is called the "captain", so the captain is involved in every product replacement.

One step in the algorithm is to do one product replacement followed by post-multiplying the result to the accumulator if one (or more) is used. Multiple accus (see the `accus` component) are used in a round robin fashion.

First `Maximum(Length(`*gens*`)*scramblefactor,scramble)` steps are performed. After this initialisation for every random element requested one step is done and the resulting element returned.

6.2.2 Next

◊ `Next(`*pr*`)` (operation)

Returns: a (pseudo-) random group element g

pr must be a product replacer object. This operation makes the object generate the next random element and return it.

6.2.3 Reset

◊ `Reset(`*pr*`)` (operation)

Returns: nothing

`pr` must be a product replacer object. This operation resets the object in the sense that it resets the product replacement back to the state it had after scrambling. Note that since the random source is not reset, the product replacer object will return another sequence of random elements than before.

6.2.4 AddGeneratorToProductReplacer

◊ `AddGeneratorToProductReplacer(`*`pr, el`*`)` (operation)

Returns: nothing

`pr` must be a product replacer object. This operation adds the new generator `el` to the product replacer without needing a completely new initialisation phase. From after this call on the product replacer will generate random elements in the group generated by the old generators and the new element `el`.

Chapter 7

Searching in groups and orbits

7.1 Searching using orbit enumeration

As described in Subsection 3.1.4 the standard orbit enumeration routines can perform search operations during orbit enumeration. If one is looking for a shortest word in the generators of a group to express a group element with a certain property, then this natural breadth-first search can be used, for example by letting the group act on its own elements, either by multiplication or by conjugation.

All technical instructions to do this are already given in Subsection 3.1.4, so we are content to provide an example here. Assume you want to find the shortest way to express some 7-cycle in the symmetric group S_{10} as a product of generators $a :=$ `(1,2,3,4,5,6,7,8,9,10)` and $b :=$ `(1,2)`. Then you could do this in the following way:

Example

```
gap> gens := [(1,2,3,4,5,6,7,8,9,10),(1,2)];
[ (1,2,3,4,5,6,7,8,9,10), (1,2) ]
gap> o := Orb(gens,(),OnRight,rec( report := 2000,
> lookingfor :=
> function(o,x) return NrMovedPoints(x) = 7 and Order(x)=7; end,
> schreier := true ));
<open orbit, 1 points with Schreier tree looking for sth.>
gap> Enumerate(o);
<open orbit, 614 points with Schreier tree looking for sth.>
gap> w := TraceSchreierTreeForward(o,PositionOfFound(o));
[ 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 2 ]
gap> ActWithWord(o!.gens,w,o!.op,o[1]);
(1,10,9,8,7,6,5)
gap> o[PositionOfFound(o)] = ActWithWord(o!.gens,w,o!.op,o[1]);
true
gap> EvaluateWord(o!.gens,w);
(1,10,9,8,7,6,5)
```

The result shows that $a^6 \cdot (a \cdot b)^3$ is a 7-cycle and that there is no word in a and b with fewer than 12 letters expressing a 7-cycle.

Note that we can go on with the above orbit enumeration to find further words to express 7-cycles.

7.2 Random searches in groups

Another possibility to look for elements in a group satisfying certain properties is to look at random elements, usually obtained by doing product replacement (see Section 6.2). Although this can lead to very long expressions as words in the generators, one can cope with this problem by using the `maxdepth` option of the product replacer objects, which just reinitialises the product replacement table after a certain number of product replacements has been performed. To organise all this conveniently, there is the concept of "random searcher objects" described here.

7.2.1 RandomSearcher

◊ `RandomSearcher(`*gens*`,` *testfunc*`[,` *opt*`])` (operation)

Returns: a random searcher object

gens must be a list of group generators, *testfunc* a function taking as argument one group element and returning `true` or `false`. *opt* is an optional options record. For possible options see below.

At first, the random searcher object is just initialised but no random searching is performed. The actual search is triggered by the `Search` (7.2.2) operation (see below). The idea of random searcher objects is that a product replacer object is created and during a search random elements are produced using this product replacer object, and for each group element produced the function *testfunc* is called. If this function returns `true`, the search stops and the group element found is returned.

The following options can be bound in the options record *opt*:

`exceptions` This component can be a list to initialise the exception list in the random searcher object. Group elements in this list are not considered as successful searches. This list is also used to continue search operations to found more suitable group elements as every group element considered "found" is added to that list before returning it. Thus every element will only be found once.

`maxdepth` Sets the `maxdepth` option of the created product replacer object. This limits the length of the expression as product of the generators of the found group elements.

`addslots` Sets the `addslots` option of the created product replacer object.

`scramble` If this component is bound at all, then the created product replacer object is created with options `scramble=100` and `scramblefactor=10` (the default values), otherwise the options `scramble=0` and `scramblefactor=0` are used, leading to no scrambling at all. See `ProductReplacer` (6.2.1) for details on the product replacement algorithm.

Note that of course the generators in *gens* may have memory. However, the function *testfunc* is called with the group element with memory stripped off.

7.2.2 Search

◊ `Search(`*rs*`)` (operation)

Returns: a group element

Runs the search with the random searcher object *rs* and returns with the first group element found.

7.3 The dihedral trick and applications

With the “dihedral” trick we mean the following: Two involutions a and b in a group always generate a dihedral group. Thus, to find a pseudo-random element in the centraliser of an involution a, we can proceed as follows: Create a pseudo-random element c, then $b := a^c$ is another involution. If then ab has order $2o$, we can use $(ab)^o$. Otherwise, if the order of ab is $2o-1$, then $(ab)^o \cdot c^{-1}$ centralises a.

This trick allows to efficiently produce elements in the centraliser of an involution and thus, with high probability, generators for the full centraliser.

There are the following functions:

7.3.1 FindInvolution

◊ `FindInvolution(`*`pr`*`)` (function)

Returns: an involution

`pr` must be a product replacer object (see Section 6.2). Searches an involution by finding a random element of even order and powering up. Returns the involution.

7.3.2 FindCentralisingElementOfInvolution

◊ `FindCentralisingElementOfInvolution(`*`pr, a`*`)` (function)

Returns: a group element

`pr` must be a product replacer object (see Section 6.2). Produces one random element and builds an element the centralises the involution *`a`* using the dihedral trick described above.

7.3.3 FindInvolutionCentralizer

◊ `FindInvolutionCentralizer(`*`pr, a, nr`*`)` (function)

Returns: a list of *`nr`* group elements

`pr` must be a product replacer object (see Section 6.2) and *`a`* and involution. This function uses `FindCentralisingElementOfInvolution` (7.3.2) *`nr`* times to produce an element centralising the involution *`a`* and returns the list of results.

7.4 Orbit statistics on vector spaces

The following two functions are tools to get a rough and quick estimate about the average orbit length of a group acting on a vector space.

7.4.1 OrbitStatisticOnVectorSpace

◊ `OrbitStatisticOnVectorSpace(`*`gens, size, ti`*`)` (function)

Returns: nothing

`gens` must be a list of matrix group generators and *`size`* an integer giving the order of the group generated by *`gens`*. *`ti`* is an integer specifying the number of seconds to run. This function enumerates orbits of random vectors in the natural space the group is acting on. The average length and some other information is printed on the screen.

7.4.2 OrbitStatisticOnVectorSpaceLines

◊ `OrbitStatisticOnVectorSpaceLines(`*`gens, size, ti`*`)` (function)

Returns: nothing

gens must be a list of matrix group generators and *size* an integer giving the order of the group generated by *gens*. *ti* is an integer specifying the number of seconds to run. This function enumerates orbits of random one-dimensional subspaces in the natural space the group is acting on. The average length and some other information is printed on the screen.

7.5 Finding generating sets of subgroups

The following function can be used to find generators of a subgroup of a group G, expressed as a straight line program in the generators of G.

7.5.1 FindShortGeneratorsOfSubgroup

◊ `FindShortGeneratorsOfSubgroup(`*`G, U[, membopt]`*`)` (method)

Returns: a record described below

The arguments *U* and *G* must be GAP group objects with *U* being a subgroup of *G*. The argument *membopt* can be a function taking two arguments, namely a group element and a group, that checks, whether the group element lies in the group or not, returning `true` or `false` accordingly. You can usually just use the function `\in` as third argument. Note that this function will only ever be called with *U* as its second argument so you can in fact provide a function which ignores its second argument and has *U* somehow built in it.

Optionally, the third argument *membopt* can also be an options record. The component `membershiptest` has the same meaning like the third argument *membopt* above. The component `sizetester` can be bound to a function which estimates the size of a group generated by some elements in *U*. This estimate function can for example be a function which runs a random Schreier-Sims algorithm. In particular it may underestimate the size with a certain probability. The method `FindShortGeneratorsOfSubgroup` will keep looking for elements to generate *U* until the `sizetester` returns the same number as for *U* itself. Note that to avoid the possibility that the `sizetester` underestimates the size of *U* in the first place you can bind the component `sizeU` in the options record to the correct size of *U* or make sure that the group object *U* does know its size before the call to `FindShortGeneratorsOfSubgroup`.

This function does a breadth-first search to find elements in *U* using the generators of *G*. It then uses calculates the size of the group generated and proceeds in this way, until a generating system for *U* is found in terms of the generators of *G*. Those generators are guaranteed to be shortest words in the generators of *G* lying in *U*.

The function returns a record with two components bound: `gens` is a list of generators for *U* and `slp` is a GAP straight line program expressing exactly those generators in the generators of *G*.

Note that this function only performs satisfactorily when the index of *U* in *G* is not to huge. It also helps if the groups come in a representation in which GAP can compute efficiently for example as permutation groups.

Chapter 8

AVL trees

8.1 The idea of AVL trees

AVL trees are balanced binary trees called “AVL trees” in honour of their inventors G.M. Adelson-Velskii and E.M. Landis (see [AVM62]). A description in English can be found in [Knu97] in Section 6.2.3 about balanced trees.

The general idea is to store data in a binary tree such that all entries in the left subtree of a node are smaller than the entry at the node and all entries in the right subtree are bigger. The tree is kept “balanced” which means that for each node the depth of the left and right subtrees differs by at most 1. In this way, finding something in the tree, adding a new entry, deleting an entry all have complexity $log(n)$ where n is the number of entries in the tree. If one additionally stores the number of entries in the left subtree of each node, then finding the k-th entry, removing the k-th entry and inserting an entry in position k also have complexity $log(n)$. The orb contains an implementation of such tree objects providing all these operations.

“Entries” in AVL tree objects are key-value pairs and the sorting is done by the key. If all values as `true` then no memory is needed to store the values (see the corresponding behaviour for hash tables). The only requirement on the type of the keys is that two arbitrary keys must be comparable in the sense that one can decide which of them is smaller. If GAPs standard comparison operations $<$ and $=$ work for your keys, no further action is required, if not, then you must provide your own three-way comparison function (see below).

Note that the AVL trees implemented here can be used in basically two different ways, which can sometimes be mixed: The usual way is by accessing entries by their key, the tree is then automatically kept sorted. The alternative way is by accessing entries by their index in the tree! Since the nodes of the trees remember how many elements are stored in their left subtree, it is in fact possible to access the k-th entry in the tree or delete it. It is even possible to insert something in position k. However, note that if you do this latter operation, you are yourself responsible to keep the entries in the tree sorted. You can ignore this responsibility, but then you can no longer access the entries in the tree by their key and the corresponding functions might fail or even run into errors.

This usage can be useful, since in this way AVL trees provide an implementation of a list data structure where the operation list access (by index), adding an element (in an arbitrary position) and deleting an element (by its index) all have complexity $log(n)$ where n is the number of entries in the list.

8.2 Using AVL trees

An AVL tree is created using the following function:

8.2.1 AVLTree

◊ `AVLTree([opt])` (function)

Returns: A new AVL tree object

This function creates a new AVL tree object. The optional argument *opt* is an options record, in which you can bind the following components:

`cmpfunc` is a three-way comparison function taking two arguments *a* and *b* and returning -1 if $a < b$, $+1$ if $a > b$ and 0 if $a = b$. If no function is given then the generic function `AVLCmp` (8.2.2) is taken. This three-way comparison function is stored with the tree and is used for all comparisons in tree operations. `allocsize` is the number of nodes which are allocated for the tree initially. It can be useful to specify this if you know that your tree will eventually contain a lot of entries, since then the tree object does not have to grow that many times.

For every AVL tree a three-way comparison function is needed, usually you can get away with using the following default one:

8.2.2 AVLCmp

◊ `AVLCmp(a, b)` (function)

Returns: -1, 0 or 1

This function calls the < operation and the = operation to provide a generic three-way comparison function to be used in AVL tree operations. See `AVLTree` (8.2.1) for a description of the return value. This function is implemented in the kernel and should be particularly fast.

The following functions are used to access entries by key:

8.2.3 AVLAdd

◊ `AVLAdd(t, key, val)` (function)

Returns: `true` or `fail`

The first argument *t* must be an AVL tree. This function stores the key *key* with value *value* in the tree assuming that the keys in it are sorted according to the three-way comparison function stored with the tree. If *value* is `true` then no additional memory is needed. It is an error if there is already a key equal to *key* in the tree, in this case the function returns `fail`. Otherwise it returns `true`.

8.2.4 AVLLookup

◊ `AVLLookup(t, key)` (function)

Returns: an value or `fail`

The first argument *t* must be an AVL tree. This function looks up the key *key* in the tree and returns the value which is associated to it. If the key is not in the tree, the value `fail` is returned. This function assumes that the keys in the tree are sorted according to the three-way comparison function stored with the tree.

8.2.5 AVLDelete

◊ AVLDelete(*t*, *key*) (function)

Returns: an value or fail

The first argument *t* must be an AVL tree. This function looks up the key *key* in the tree, deletes it and returns the value which was associated with it. If *key* is not contained in the tree then fail is returned. This function assumes that the keys in the tree are sorted according to the three-way comparison function stored with the tree.

8.2.6 AVLFindIndex

◊ AVLFindIndex(*t*, *key*) (function)

Returns: an integer or fail

The first argument *t* must be an AVL tree. This function looks up the key *key* in the tree and returns the index, under which it is stored in the tree. This index is one-based, that is, it takes values from 1 to the number of entries in the tree. If *key* is not contained in the tree then fail is returned. This function assumes that the keys in the tree are sorted according to the three-way comparison function stored with the tree.

The following functions are used to access entries in trees by their index:

8.2.7 AVLIndex

◊ AVLIndex(*t*, *index*) (function)

Returns: a key or fail

The first argument *t* must be an AVL tree. This function returns the key at index *index* in the tree, so *index* must be an integer in the range 1 to the number of elements in the tree. If the value is out of these bounds, fail is returned. Note that to use this function it is not necessary that the keys in the tree are sorted according to the three-way comparison function stored with the tree.

8.2.8 AVLIndexLookup

◊ AVLIndexLookup(*t*, *index*) (function)

Returns: a value or fail

The first argument *t* must be an AVL tree. This function returns the value associated to the key at index *index* in the tree, so *index* must be an integer in the range 1 to the number of elements in the tree. If the value is out of these bounds, fail is returned. Note that to use this function it is not necessary that the keys in the tree are sorted according to the three-way comparison function stored with the tree.

8.2.9 AVLIndexAdd

◊ AVLIndexAdd(*t*, *key*, *value*, *index*) (function)

Returns: a key or fail

The first argument *t* must be an AVL tree. This function inserts the key *key* at index *index* in the tree and associates the value *value* with it. If *value* is true then no additional memory is needed to store the value. The index *index* must be an integer in the range 1 to $n+1$ where n is the number of entries in the tree. The new key is inserted before the key which currently is stored at index

index, so calling with *index* equal to $n+1$ puts the new key at the end. If *index* is not in the corrent range, this function returns `fail` and the tree remains unchanged.

CAUTION: With this function it is possible to put a key into the tree at a position such that the keys in the tree are no longer sorted according to the three-way comparison function stored with the tree! If you do this, the functions `AVLAdd` (8.2.3), `AVLLookup` (8.2.4), `AVLDelete` (8.2.5) and `AVLFindIndex` (8.2.6) will no longer work since they assume that the keys are sorted!

8.2.10 AVLIndexDelete

◊ `AVLIndexDelete(`*t*`, `*index*`)` (function)

Returns: a key or `fail`

The first argument *t* must be an AVL tree. This function deletes the key at index *index* in the tree and returns the value which was associated with it.

The following functions allow low level access to the AVL tree object:

8.2.11 AVLFind

◊ `AVLFind(`*t*`, `*key*`)` (function)

Returns: an integer or `fail`

The first argument *t* must be an AVL tree. This function locates the key *key* in the tree and returns the position in the positional object, at which the node which contains the key is stored. This position will always be divisible by 4. Use the functions `AVLData` (8.2.13) and `AVLValue` (8.2.14) to access the key and value of the node respectively. The function returns `fail` if the key is not found in the tree. This function assumes that the keys in the tree are sorted according to the three-way comparison function stored with the tree.

8.2.12 AVLIndexFind

◊ `AVLIndexFind(`*t*`, `*index*`)` (function)

Returns: an integer or `fail`

The first argument *t* must be an AVL tree. This function locates the index *index* in the tree and returns the position in the positional object, at which the node which hash this index is stored. This position will always be divisible by 4. Use the functions `AVLData` (8.2.13) and `AVLValue` (8.2.14) to access the key and value of the node respectively. The function returns `fail` if the key is not found in the tree. This function does not assume that the keys in the tree are sorted according to the three-way comparison function stored with the tree.

8.2.13 AVLData

◊ `AVLData(`*t*`, `*pos*`)` (function)

Returns: an key

The first argument *t* must be an AVL tree and the second a position in the positional object corresponding to a node as returned by `AVLFind` (8.2.11). The function returns the key associated with this node.

8.2.14 AVLValue

◊ `AVLValue(t, pos)` (function)

Returns: a value

The first argument *t* must be an AVL tree and the second a position in the positional object corresponding to a node as returned by `AVLFind` (8.2.11). The function returns the value associated with this node.

The following convenience methods for standard list methods are implemented for AVL tree objects:

8.2.15 Display

◊ `Display(t)` (method)

Returns: nothing

This function displays the tree in a user-friendly way. Do not try this with trees containing many nodes!

8.2.16 ELM_LIST

◊ `ELM_LIST(t, index)` (method)

Returns: A key or `fail`

This method allows for easy access to the key at index *index* in the tree using the square bracket notation `t[index]`. It does exactly the same as `AVLIndex` (8.2.7). This is to make AVL trees behave more like lists.

8.2.17 Position

◊ `Position(t, key)` (method)

Returns: an integer or `fail`

This method allows to use the `Position` operation to locate the index at which the key *key* is stored in the tree. It does exactly the same as `AVLFindIndex` (8.2.6). This is to make AVL trees behave more like lists.

8.2.18 Add

◊ `Add(t, key[, index])` (method)

Returns: nothing

This method allows to use the `Add` operation to add a key (with associated value `true`) to the tree at index *index*. It does exactly the same as `AVLIndexAdd` (8.2.9), so the same warning about sortedness as there applies! If *index* is omitted, the key is added at the end. This is to make AVL trees behave more like lists.

8.2.19 Remove

◊ `Remove(t, index)` (method)

Returns: a key

This method allows to use the `Remove` operation to remove a key from the tree at index *index*. If *index* is omitted, the last key in the tree is remove. This method returns the deleted key or `fail` if the tree was empty. This is to make AVL trees behave more like lists.

8.2.20 Length

◊ `Length(t)` (method)

Returns: a key

This method returns the number of entries stored in the tree *t*. This is to make AVL trees behave more like lists.

8.2.21 \in

◊ `\in(key, t)` (method)

Returns: `true` or `false`

This method tests whether or not the key *key* is stored in the AVL tree *t*. This is to make AVL trees behave more like lists.

8.3 The internal data structures

An AVL tree is a positional object in which the first 7 positions are used for administrative data (see table below) and then from position 8 on follow the nodes of the tree. Each node uses 4 positions such that all nodes begin at positions divisible by 4. The system allocates the positional object larger than actually needed such that not every new node leads to the object being copied. Nodes which become free are collected in a free list. The following table contains the information what is stored in each of the first 7 entries:

1	last actually used position, is always congruent 3 mod 4
2	position of first node in free list
3	number of currently used nodes in the tree
4	position of largest allocated position is always congruent 3 mod 4
5	three-way comparison function
6	position of the top node
7	a plain list holding the values stored under the keys

The four positions used for a node contain the following information, recall that each node starts at a position divisible by 4:

0 mod 4	reference to the key
1 mod 4	position of left node or 0 if empty, balance factor (see below)
2 mod 4	position of right node or 0 if empty
3 mod 4	index: number of nodes in left subtree plus one

Since all positions of nodes are divisible by 4, we can use the least significant two bits of the left node reference for the so called balance factor. Balance factor 0 (both bits 0) indicates that the depth of the left subtree is equal to the depth of the right subtree. Balance factor 1 (bits 01) indicates that the depth of the right subtree is one greater than the depth of the left subtree. Balance factor 2 (or -1 in [Knu97], here bits 10) indicates that the depth of the left subtree is one greater than the depth of the right subtree.

For freed nodes the position of the next free node in the free list is held in the 0 mod 4 position and 0 means the end of the free list.

Position 7 in the positional object can contain the value `fail`, in this case all stored values are `true`. This is a measure to limit the memory usage in the case that the only relevant information in the tree is the key and no values are stored there. This is in particular interesting if the tree structure is just used as a list implementation.

Note that all functions dealing with AVL trees are both implemented on the GAP level and on the kernel level. Both implementations do exactly the same thing, the kernel version is only much faster and tuned for efficiency whereas the GAP version documents the functionality better and is used as a fallback if the C-part of the orb is not compiled.

Chapter 9

Orbit enumeration by suborbits

The code described in this chapter is quite complicated and one has to understand quite a lot of theory to use it. The reason for this is that a lot of preparatory data has to be found and supplied by the user in order for this code to run at all. Also the situations in which it can be used are quite special. However, in such a situation, the user is rewarded with impressive performance.

The main reference for the theory is [MNW07]. We briefly recall the basic setup: Let G be a group acting from the right on some set X. Let k be a natural number, set $X_{k+1} := X$, and let

$$U_1 < U_2 < \ldots < U_k < U_{k+1} = G$$

be a chain of "helper" subgroups. Further, for $1 \leq i \leq k$ let X_i be a U_i set and let $\pi_i : X_{i+1} \to X_i$ be a homomorphism of U_i-sets.

This chapter starts with a section about the main orbit enumeration function and the corresponding preparation functions. It then proceeds with a section on the used data structures, which will necessarily be rather technical. Finally, the chapter concludes with a section on higher level data structures like lists of orbit-by-suborbit objects and their administration. Note that there are quite a few examples in Chapter 11.

9.1 `OrbitBySuborbits` and its resulting objects

9.1.1 OrbitBySuborbit

◊ `OrbitBySuborbit(`*`setup, p, j, l, i, percentage`*`)` (function)

Returns: an orbit-by-suborbit object

This is the main function in the whole business. All notations from the beginning of this Chapter 9 remain in place. The argument *`setup`* must be a setup record lying in the filter `IsOrbitBySuborbitSetup` (9.3.1) described in detail in Section 9.3 and produced for example by `OrbitBySuborbitBootstrapForVectors` (9.2.1) or `OrbitBySuborbitBootstrapForLines` (9.2.2) described below. In particular, it contains all the generators for G and the helper subgroups acting on the various sets. The argument *`p`* must be the starting point of the orbit. Note that the function possibly does not take *`p`* itself as starting point but rather its U_k-minimalisation, which is a point in the same U_k-orbit as *`p`*. This information is important for the resulting stabiliser and words representing the U_k-suborbits.

The integers *`j`*, *`l`*, and *`i`*, for which $k+1 \geq$ *`j`* $\geq$ *`l`* $>$ *`i`* ≥ 1 must hold, determine the running mode. *`j`* indicates in which set X_j the point *`p`* lies and thus in which set the orbit enumeration takes place, with $j = k+1$ indicating the original set X. The value *l* indicates which group to use for orbit enumeration. So the result will be a U_l orbit, with $l = k+1$ indicating a G-orbit. Finally, the value *i* indicates which group to use for the "by suborbit" part, that is, the orbit will be enumerated "by U_i-orbits". Note that nearly all possible combinations of these parameters actually occur, because this function is also used in the "on-the-fly" precomputation happening behind the scenes. The most common usage of this function for the user is $j = l = k+1$ and $i = k$.

Finally, the integer *percentage* says, how much of the full orbit should be enumerated, the value is in percent, thus 100 means the full orbit. Usually, only values greater than 50 are sensible, because one can only prove the size of the orbit after enumerating at least half of it.

The result is an "orbit-by-suborbit" object. For such an object in particular the operations `Size` (9.1.3), `Seed` (9.1.4), `SuborbitsDb` (9.1.5), `WordsToSuborbits` (9.1.6), `Memory` (9.1.7), `Stabilizer` (9.1.8), and `Seed` (9.1.4) are defined, see below.

9.1.2 OrbitBySuborbitKnownSize

◊ `OrbitBySuborbitKnownSize(`*setup, p, j, l, i, percentage, knownsize*`)` (function)

Returns: an orbit-by-suborbit object

Basically does the same as `OrbitBySuborbit` (9.1.1) but does not compute the stabiliser by evaluating Schreier words. Instead, the size of the orbit to enumerate must already be known and be given in the argument *knownsize*. The other arguments are as for the function `OrbitBySuborbit` (9.1.1).

9.1.3 Size (fororb)

◊ `Size(`*orb*`)` (method)

Returns: an integer

Returns the number of points in the orbit-by-suborbit *orb*.

9.1.4 Seed

◊ `Seed(`*orb*`)` (method)

Returns: a point in the orbit

Returns the starting point of the orbit-by-suborbit *orb*. It is the U_i-minimalisation of the starting point given to `OrbitBySuborbit` (9.1.1).

9.1.5 SuborbitsDb

◊ `SuborbitsDb(`*orb*`)` (operation)

Returns: a database of suborbits

Returns the data base of suborbits of the orbit-by-suborbit object *orb*. In particular, such a database object has methods for the operations `Memory` (9.1.7), `TotalLength` (9.1.11), and `Representatives` (9.1.12). For descriptions see below.

9.1.6 WordsToSuborbits

◊ `WordsToSuborbits(`*orb*`)` (operation)

Returns: a list of words

Returns a list of words in the groups U_* reaching each of the suborbits in the orbit-by-suborbit *orb*. Here a word is a list of integers. Positive numbers index generators in following numbering: The first few numbers are numbers of generators of U_1 the next few adjacent numbers index the generators of U_2 and so on until the generators of G in the end. Negative numbers indicate the corresponding inverses of these generators.

Note that `OrbitBySuborbit` (9.1.1) takes the U_i-minimalisation of the starting point as its starting point and the words here are all relative to this new starting point.

9.1.7 Memory (forob)

◇ `Memory(`*ob*`)` (operation)

Returns: an integer

Returns the amount of memory needed by the object *ob*, which can be either an orbit-by-suborbit object, a suborbit database object, or an object in the filter `IsOrbitBySuborbitSetup` (9.3.1). The amount of memory used is given in bytes. Note that this includes all hashes, databases, and preparatory data of substantial size. For orbit-by-suborbits the memory needed for the precomputation is not included, ask the setup object for that.

9.1.8 Stabilizer (obso)

◇ `Stabilizer(`*orb*`)` (method)

Returns: a permutation group

Returns the stabiliser of the starting point of the orbit-by-suborbit in *orb* in form of a permutation group, using the given faithful permutation representation in the setup record.

Note that `OrbitBySuborbit` (9.1.1) takes the U_i-minimalisation of the starting point as its starting point and the stabiliser returned here is the one of this new starting point.

9.1.9 StabWords

◇ `StabWords(`*orb*`)` (operation)

Returns: a list of words

Returns generators for the stabiliser of the starting point of the orbit-by-suborbit in *orb* in form of words as described with the operation `WordsToSuborbits` (9.1.6). Note again that `OrbitBySuborbit` (9.1.1) takes the U_i-minimalisation of the starting point as its starting point and the stabiliser returned here is the one of this new starting point.

9.1.10 SavingFactor (fororb)

◇ `SavingFactor(`*orb*`)` (operation)

Returns: an integer

Returns the quotient of the total number of points stored in the orbit-by-suborbit *orb* and the total number of U-minimal points stored. Note that the memory for the precomputations is not considered here!

The following operations apply to orbit-by-suborbit database objects:

9.1.11 TotalLength (fordb)

◊ `TotalLength(db)` (operation)

Returns: an integer

Returns the total number of points stored in all suborbits in the orbit-by-suborbit database *db*.

9.1.12 Representatives

◊ `Representatives(db)` (operation)

Returns: a list of points

Returns a list of representatives of the suborbits stored in the orbit-by-suborbit database *db*.

9.1.13 SavingFactor (fordb)

◊ `SavingFactor(db)` (operation)

Returns: an integer

Returns the quotient of the total number of points stored in the suborbit database *db* and the total number of U-minimal points stored. Note that the memory for the precomputations is not considered here!

9.1.14 OrigSeed

◊ `OrigSeed(orb)` (operation)

Returns: a point

Returns the original starting point for the orbit, not yet minimalised.

9.2 Preparation functions for `OrbitBySuborbit` (9.1.1)

9.2.1 OrbitBySuborbitBootstrapForVectors

◊ `OrbitBySuborbitBootstrapForVectors(gens, permgens, sizes, codims, opt)` (function)

Returns: a setup record in the filter `IsOrbitBySuborbitSetup` (9.3.1)

All notations from the beginning of this Chapter 9 remain in place. This function is for the action of matrices on row vectors, so all generators must be matrices. The set X thus is a row space usually over a finite field and the sets X_i are quotient spaces. The matrix generators for the various groups have to be adjusted with a base change, such that the canonical projection onto X_i is just to take the first few entries in a vector, which means, that the submodules divided out are generated by the last standard basis vectors.

The first argument *gens* must be a list of lists of generators. The outer list must have length $k+1$ with entry i being a list of matrices generating U_i, given in the action on $X = X_{k+1}$. The above mentioned base change must have been done. The second argument *permgens* must be an analogous list with generator lists for the U_i. These representations are used to compute membership and group orders of stabilisers. In its simplest form, *permgens* is a list of permutation representations of the same degree, giving a set of generators for each individual group U_i. Alternatively, if for some U_i, $i > 1$, it is required that the stabilizer of its action is to be calculated as a matrix group, generators of U_i in some matrix representation may be supplied. However, it is then mandatory that for all $1 < i \leq k+1$ the generator lists have the following format: The i-th entry of *permgens* is a list concatenating the generator lists of U_1 up to U_i (in this order) all of whose elements are in either some permutation or some matrix representation. Note that currently, the generators of U_1 need to be always given in a permutation representation. The argument *sizes* must be a list of length $k+1$ and entry i must be the group order of U_i (again with U_{k+1} being G). Finally, the argument *codims* must be a list of length k containing integers with the ith entry being the codimension of the U_i-invariant subspace Y_i of X with $X_i = X/Y_i$. These codimensions must not decrease for obvious reasons, but some of them may be equal. The last argument *opt* is an options record. See below for possible entries.

The function does all necessary steps to fill a setup record (see 9.3) to be used with `OrbitBySuborbit` (9.1.1). For details see the code.

Currently, the following components in the options record *opt* have a meaning:

regvecfachints If bound it must be a list. In position i for $i > 1$ there may be a list of vectors in the i-th quotient space X_i that can be used to distinguish the left U_{i-1} cosets in U_i. All vectors in this list are tried and the first one that actually works is used.

regvecfullhints If bound it must be a list. In position i for $i > 1$ there may be a list of vectors in the full space X that can be used to distinguish the left U_{i-1} cosets in U_i. All vectors in this list are tried and the first one that actually works is used.

stabchainrandom If bound the value is copied into the `stabchainrandom` component of the setup record.

nostabchainfullgroup If bound it must be `true` or `false`. If it is unbound or set to `true`, no stabilizer chain is computed for the group U_{k+1}. Its default value is `false`.

9.2.2 OrbitBySuborbitBootstrapForLines

◊ `OrbitBySuborbitBootstrapForLines(`*gens, permgens, sizes, codims, opt*`)` (function)

Returns: a setup record in the filter `IsOrbitBySuborbitSetup` (9.3.1)

All notations from the beginning of this Chapter 9 remain in place. This does exactly the same as `OrbitBySuborbitBootstrapForVectors` (9.2.1) except that it handles the case of matrices acting on one-dimensional subspaces. Those one-dimensional subspaces are represented by normalised vectors, where a vector is normalised if its first non-vanishing entry is equal to 1.

9.2.3 OrbitBySuborbitBootstrapForSpaces

◊ `OrbitBySuborbitBootstrapForSpaces(`*gens, permgens, sizes, codims, spcdim, opt*`)` (function)

Returns: a setup record in the filter `IsOrbitBySuborbitSetup` (9.3.1)

All notations from the beginning of this Chapter 9 remain in place. This does exactly the same as `OrbitBySuborbitBootstrapForVectors` (9.2.1) except that it handles the case of matrices acting on *spcdim*-dimensional subspaces. Those subspaces are represented by fully echelonised bases.

9.3 Data structures for orbit-by-suborbits

The description in this section is necessarily technical. It is meant more as extended annotations to the source code than as user documentation. Usually it should not be necessary for the user to know the details presented here. The function `OrbitBySuborbit` (9.1.1) needs an information record of the following form:

9.3.1 IsOrbitBySuborbitSetup

◊ `IsOrbitBySuborbitSetup(ob)` (Category)

Returns: `true` or `false`

Objects in this category are also in `IsComponentObjRep`. We describe the components, refering to the setup at the beginning of this Chapter 9.

`k` The number of helper subgroups.

`size` A list of length $k+1$ containing the orders of the groups U_i, including $U_{k+1} = G$.

`index` A list of length k with the index $[U_i : U_{i-1}]$ in position i ($U_0 = \{1\}$).

`els` A list of length $k+1$ containing generators of the groups in their action on various sets. In position i we store all the generators for all groups acting on X_i, that is for the groups $U_1, \ldots, U_i$ (where position $k+1$ includes the generators for G. In each position the generators of all those groups are concatentated starting with U_1 and ending with U_i.

`elsinv` The inverses of all the elements in the `els` component in the same arrangement.

`trans` A list of length k in which position i for $i > 1$ contains a list of words in the generators for a transversal of U_{i-1} in U_i (with $U_0 = 1$).

`pifunc` Projection functions. This is a list of length $k+1$ containing in position j a list of length $j-1$ containing in position i a GAP function doing the projection $X_j \to X_i$. These GAP functions take two arguments, namely the point to map and secondly the value of the `pi` component at positions `[j][i]`. Usually `pifunc` is just the slicing operator in GAP and `pi` contains the components to project onto as a range object.

`pi` See the description of the `pifunc` component.

`op` A list of $k+1$ GAP operation functions, each taking a point p and a generator g in the action given by the index and returning pg.

`info` A list of length k containing a hash table with the minimalisation lookup data. These hash tables grow during orbit enumerations as precomputations are done behind the scenes.

`info[1]` contains precomputation data for X_1. Assume $x \in X_1$ to be U_1-minimal. For all $z \in xU_1$ with $z \neq x$ we store the number of an element in the `wordcache` mapping z to x. For $z = x$ we store a record with two components `gens` and `size`, where `gens` stores generators for the stabiliser $\mathrm{Stab}_{U_1}(x)$ as words in the group generators and `size` stores the size of that stabiliser.

`info[i]` for $i > 1$ contains precomputation data for X_i. Assume $x \in X_i$ to be U_i-minimal. For all U_{i-1}-minimal $z \in xU_i \setminus xU_{i-1}$ we store the number of an element in `trans[i]` mapping z into xU_{i-1}. For all U_{i-1}-minimal $z \in xU_{i-1}$ with $z \neq x$ we store the negative of the number of a word in `wordcache` that is in the generators of U_{i-1} and maps z to x. For $z = x$ we store the stabiliser information as in the case $i = 1$.

This information together with the information in the following componente allows the minimalisation function to do its job.

cosetrecog A list of length k beginning with the index 1. The entry at position i is bound to a function taking 3 arguments, namely i itself, a word in the group generators of $U_1,\ldots,U_k$ which lies in U_i, and the setup record. The function computes the number `j` of an element in `trans[i]`, such that the element of U_i described by the word lies in `trans[i][j] U_{{i-1}}`.

cosetinfo A list of things that can be used by the functions in `cosetrecog`.

suborbnr A list of length k that contains in position i the number of U_i-orbits in X_i archived in `info[i]` during precomputation.

sumstabl A list of length k that contains in position i the sum of the point stabiliser sizes of all U_i-orbits X_i archived in `info[i]` during precomputation.

permgens A list of length $k+1$ containing in position i generators for $U_1,\ldots,U_i$ in a faithful permutation representation of U_i. Generators fit to the generators in `els`. For the variant `OrbitBySuborbitKnownSize` (9.1.2) the $k+1$ entry can be unbound.

permgensinv The inverses of the generators in `permgens` in the same arrangement.

sample A list of length $k+1$ containing sample points in the sets X_i.

stabchainrandom The value is used as the value for the `random` option for `StabChain` calculations to determine stabiliser sizes. Note that the algorithms are randomized if you use this feature with a value smaller than 1000.

wordhash A hash to quickly recognise already used words. For every word in the hash the position of that word in the `wordcache` list is stored as value in the hash.

wordcache A list of words in the wordcache for indexing purposes.

hashlen Initial length of hash tables used for the enumeration of lists of U_i-minimal points.

staborblenlimit This contains the limit, up to which orbits of stabilisers are computed using word action. After this limit, the stabiliser elements are actually evaluated in the group.

stabsizelimitnostore If the stabiliser in the quotient is larger than this limit, the suborbit is not stored.

cache A linked list cache object (see `LinkedListCache` (5.2.1)) to store already computed transversal elements. The cache nodes are referenced in the `transcache` component and are stored in the cache `cache`.

transcache This is a list of lists of weak pointer objects. The weak pointer object at position `[i][j]` holds references to cache nodes of transversal elements of U_{i-1} in U_i in representation j.

9.3.2 The global record `ORB`

In this section we describe the global record `ORB`, which contains some entries that can tune the behaviour of the orbit-by-suborbit functions. The record has the following components:

MINSHASHLEN This positive integer is the initial value of the hash size when enumerating orbits of stored stabilisers to find all or search through U_{i-1}-minimal vectors in an U_i-orbit. The default value is 1000.

ORBITBYSUBORBITDEPTH This integer indicates how many recursive calls to `OrbitBySubOrbitInner` have been done. The initial value is 0 to indicate that no such call has happened. This variable is necessary since the minimalisation routine sometimes uses `OrbitBySubOrbitInner` recursively to complete some precomputation "on the fly" during some other orbit-by-suborbit enumeration. This component is always set to 0 automatically when calling `OrbitBySuborbit` (9.1.1) or `OrbitBySuborbitKnownSize` (9.1.2) so the user should usually not have to worry about it at all.

PATIENCEFORSTAB This integer indicates how many Schreier generators for the stabiliser are tried before assuming that the stabiliser is complete. Whenever a new generator for the stabiliser is found that increases the size of the currently known stabiliser, the count is reset to 0 that is, only when `ORB.PATIENCEFORSTAB` unsuccessful Schreier generators have been tried no more Schreier generators are created. The default value for this component is 1000. This feature is purely heuristical and therefore this value has to be adjusted for some orbit enumerations.

PLEASEEXITNOW This value is usually set to `false`. Setting it to `true` in a break loop tells the orbit-by-suborbit routines to exit gracefully at the next possible time. Simply leaving such a break loop with `quit;` is not safe, since the routines might be in the process of updating precomputation data and the data structures might be left corrupt. Always use this component to leave an orbit enumeration prematurely.

REPORTSUBORBITS This positive integer governs how often information messages about newly found suborbits are printed. The default value is 1000 saying that after every 1000 suborbits a message is printed, if the info level is at its default value 1. If the info level is increased, then this component does no longer affect the printing and all found suborbits are reported.

TRIESINQUOTIENT and TRIESINWHOLESPACE The bootstrap routines `OrbitBySuborbitBootstrapForVectors` (9.2.1), `OrbitBySuborbitBootstrapForLines` (9.2.2) and `OrbitBySuborbitBootstrapForSpaces` (9.2.3) all need to compute transversals of one helper subgroup in the next one. They use orbit enumerations in various spaces to achieve this. The component `TRIESINQUOTIENT` must be a non-negative integer and indicates how often a random vector in the corresponding quotient space is tried to find an orbit that can distinguish between cosets. The other component `TRIESINWHOLESPACE` also must be a non-negative integer and indicates how often a random vector in the whole space is tried. The default values are 3 and 20 resepectively.

9.4 Lists of orbit-by-suborbit objects

There are a few functions that help to administrate lists of orbit-by-suborbits.

9.4.1 InitOrbitBySuborbitList

◊ `InitOrbitBySuborbitList(`*`setup, nrrandels`*`)` (function)

Returns: a list of orbit-by-suborbits object

Creates an object that stores a list of orbit-by-suborbits. The argument *setup* must be an orbit-by-suborbit setup record and *nrrandels* must be an integer. It indicates how many random elements in G should be used to do a probabilistic check for membership in case an orbit-by-suborbit is only partially known.

9.4.2 IsVectorInOrbitBySuborbitList

◊ `IsVectorInOrbitBySuborbitList(`*v, obsol*`)` (function)

Returns: `fail` or an integer

Checks probabilistically, if the element *v* lies in one of the partially enumerated orbit-by-suborbits in the orbit-by-suborbit list object *obsol*. If yes, the number of that orbit-by-suborbit is returned and the answer is guaranteed to be correct. If the answer is `fail` there is a small probability that the point actually lies in one of the orbits but this could not be shown.

9.4.3 OrbitsFromSeedsToOrbitList

◊ `OrbitsFromSeedsToOrbitList(`*obsol, li*`)` (function)

Returns: nothing

Takes the elements in the list *li* as seeds for orbit-by-suborbits. For each such seed it is first checked whether it lies in one of the orbit-by-suborbits in *obsol*, which must be an orbit-by-suborbit list object. If not found, 51% of the orbit-by-suborbit of the seed is enumerated and added to the list *obsol*.

This function is a good way to quickly enumerate a greater number of orbit-by-suborbits.

9.4.4 VerifyDisjointness

◊ `VerifyDisjointness(`*obsol*`)` (function)

Returns: `true` or `false`

This function checks deterministically, whether the orbit-by-suborbits in the orbit-by-suborbit list object *obsol* are disjoint or not and returns the corresponding boolean value. This is not a Monte-Carlo algorithm. If the answer is `false`, the function writes out, which orbits are in fact identical.

9.4.5 Memory (forobsol)

◊ `Memory(`*obsol*`)` (operation)

Returns: an integer

Returns the total memory used for all orbit-by-suborbits in the orbit-by-suborbit-list *obsol*. Precomputation data is not included, ask the setup object instead.

9.4.6 TotalLength (forobsol)

◊ `TotalLength(`*obsol*`)` (operation)

Returns: an integer

Returns the total number of points stored in all orbit-by-suborbits in the orbit-by-suborbit-list *obsol*.

9.4.7 Size (forobsol)

◊ `Size(obsol)` (method)

Returns: an integer

Returns the total number of points in the orbit-by-suborbit-list `obsol`.

9.4.8 SavingFactor (forobsol)

◊ `SavingFactor(obsol)` (operation)

Returns: an integer

Returns the quotient of the total number of points stored in all orbit-by-suborbits in the orbit-by-suborbit-list `obsol` and the total number of U-minimal points stored, which is the average saving factor considering all orbit-by-suborbits together. Note that the memory for the precomputations is not considered here!

Chapter 10

Finding nice quotients

This chapter will be written when the chop is documented and released, because the functions to be described here depend on that package.

For the moment it should be enough to say that the functions to be described here are used to find nice quotient modules for the orbit algorithms using the orbit-by-suborbit techniques described in Chapter 9.

Chapter 11

Examples

To actually run an orbit enumeration by suborbits, we have to collect some insight into the structure of the group under consideration and into its representation theory. In general, preparing the input data is more of an art than a science. The mathematical details are described in [MNW07].

In Section 11.1 we present a small example of the usage of the orbit-by-suborbit machinery. We use the sporadic simple Mathieu group M_{11} acting projectively on its irreducible module of dimension 24 over the field with 3 elements.

In Section 11.2 we present another example of the usage of the orbit-by-suborbit programs. In this example we determine 35 of the 36 double coset representatives of the sporadic simple Fischer group Fi_{23} with respect to its seventh maximal subgroup.

In Section 11.3 we present a bigger example of the usage of the orbit-by-suborbit machinery. In this example the orbit lengths of the sporadic simple Conway group Co_1 acting in in its irreducible projective representation over the field with 5 elements in dimension 24 are determined, which were previously unknown. These orbit lengths were needed to rule out a case in [Mal06].

In Section 11.4 we present as an extended worked example how to enumerate the smallest non-trivial orbit of the sporadic simple Baby Monster group B. We give a log of a GAP session with explanations in between, being intended to illustrate a few of the tools which are available in the orb package as well as in related packages. Actually, the orb package has also been applied to two much larger permutation actions of B, namely its action on its 2B involutions, having degree $\approx 1.2 \cdot 10^{13}$, and its action on the cosets of a maximal subgroup isomorphic to Fi_{23}, having degree $\approx 1.0 \cdot 10^{15}$; for details see [Mül08] and [MNW07], respectively.

Note that for all this to work you have to acquire and install the packages IO, cvec, and atlasrep, and for Section 11.4 you additionally need the packages chop and genss.

11.1 The Mathieu group M_{11} acting in dimension 24

The example in this section is very small but our intention is that everything can still be analysed and looked at more or less by hand. We want to enumerate orbits of the Mathieu group M_{11} acting projectively on its irreducible module of dimension 24 over the field with 3 elements. All the files for this example are located in the `examples/m11PF3d24` subdirectory of the orb package. Then you simply run the example in the following way:

Example

```
gap> ReadPackage("orb","examples/m11PF3d24/M11OrbitOnPF3d24.g");
...
gap> o := OrbitBySuborbit(setup,v,3,3,2,100);
```

```
...
#I  OrbitBySuborbit found 100% of a U3-orbit of size 7 920
...
```

Everything works instantly as it would have without the orbit-by-suborbits method. (Depending on whether the matrix and permutation generators for M_{11} are already stored locally, some time might be needed to fetch them.) The details of this computation can be directly read off from the code in the file `M11OrbitOnPF3d24.g`:

Example

```
LoadPackage("orb");
LoadPackage("io");
LoadPackage("cvec");
LoadPackage("atlasrep");

SetInfoLevel(InfoOrb,2);
pgens := AtlasGenerators("M11",1).generators;

gens := AtlasGenerators("M11",14).generators;
cgens := List(gens,CMat);
basech := CVEC_ReadMatFromFile(Filename(DirectoriesPackageLibrary("orb",""),
          "examples/m11PF3d24/m11basech.cmat"));
basechi := basech^-1;
cgens := List(cgens,x->basech*x*basechi);

ReadPackage("orb","examples/m11PF3d24/m11slps.g");
pgu2 := ResultOfStraightLineProgram(s2,pgens);
pgu1 := ResultOfStraightLineProgram(s1,pgu2);
cu2 := ResultOfStraightLineProgram(s2,cgens);
cu1 := ResultOfStraightLineProgram(s1,cu2);

setup := OrbitBySuborbitBootstrapForLines(
         [cu1,cu2,cgens],[pgu1,pgu2,pgens],[20,720,7920],[5,11],rec());
setup!.stabchainrandom := 900;

v := ZeroMutable(cgens[1][1]);
Randomize(v);
ORB_NormalizeVector(v);

Print("Now do\n  o := OrbitBySuborbit(setup,v,3,3,2,100);\n");
```

We are using two helper subgroups $U_1 < U_2 < M_{11}$, where $U_2 \cong A_2.2$ is the largest maximal subgroup of M_{11}, having order 720, and $U_2 \cong 5:4$ is a maximal subgroup of U_2 of order 20, see [CCN+85] or the CTblLib package. The quotient spaces we use for the helper subgroups have dimensions 5 and 11 respectively. Straight line programs to compute generators of the helper subgroups in terms of the given generators of M_{11}, and an appropriate basis exhibiting the quotients, have already been computed, and are stored in the files `m11slps.g` and `m11basech.cmat`, respectively. (In Section 11.4 we show in detail how such straight line programs and suitable bases can be found using the tools available in in the orb package.) The command `OrbitBySuborbitBootstrapForLines` invokes the precomputation, and in particular says that we want to use projective action.

11.2 The Fischer group Fi_{23} acting in dimension 1494

The example in this section shows how to compute 35 of the 36 double coset representatives of the Fischer group Fi_{23} with respect to its the seventh maximal subgroup $H \cong 3_+^{1+8}.2_-^{1+6}.3_+^{1+2}.2S_4$, which has order $3\,265\,173\,504 \approx 3.2 \cdot 10^9$ and index $[Fi_{23}:H] = 1\,252\,451\,200 \approx 1.3 \cdot 10^9$, see [CCN+85] or the CTblLib package. All the files for this example are located in the `examples/fi23m7` subdirectory of the orb package. You simply run the example in the following way:

Example

```
gap> ReadPackage("orb","examples/fi23m7/GOrbitByKOrbitsPrepare.g");
...
gap> ReadPackage("orb","examples/fi23m7/GOrbitByKOrbitsSearch35.g");
...
```

We will not go into the details of the computation here, but they can be read off directly from the code in the files in that directory. In the first part, run by the file `GOrbitByKOrbitsPrepare.g`, we prepare the necessary input data, by using similar techniques as described at length in Section 11.4. (Actually, this example has been dealt with before the advent of the packages chop and genss, hence we are using appropriate private code instead.) We are using two helper subgroups $U_1 < U_2 < H < Fi_{23}$, being 3-subgroups of H of order 81 and 6561, respectively. The 1494-dimensional irreducible representation of Fi_{23} over the field with 2 elements contains a vector that is fixed by H, such that the action on its Fi_{23}-orbit is isomorphic to the action on the cosets of H.

The second part, in the file `GOrbitByKOrbitsSearch35.g`, is the actual enumeration of H-orbits:

Example

```
setup := OrbitBySuborbitBootstrapForVectors(
         [cu1gens,cu2gens,cngens],[u1gensp,u2gensp,ngensp],
         [81,6561,3265173504],[10,30],rec());
obsol := InitOrbitBySuborbitList(setup,40);
l := Orb(cggens,v,OnRight,rec(schreier := true));
Enumerate(l,100000);
OrbitsFromSeedsToOrbitList(obsol,l);
origseeds := List(obsol,OrigSeed);
positions :=  List(origseeds,x->Position(l,x));
words := List(positions,x->TraceSchreierTreeForward(l,x));
```

Note that this computation finds only 35 of the 36 double coset representatives. The last corresponds to a very short suborbit which is very difficult to find. Knowing the number of missing points, we guess the stabiliser in H of a missing representative, and find the latter amongst the fixed points of the stabiliser. We can then choose the one which lies in the G-orbit we have nearly enumerated above.

These double coset representatives were needed to determine the 2-modular character table of Fi_{23}. Details of this can be found in [HNN06].

11.3 The Conway group Co_1 acting in dimension 24

The example in this section shows how to compute all suborbit lengths of the Conway group Co_1, in its irreducible projective action on a module of dimension 24 over the field with 5 elements. All the files for this example are located in the `examples/co1F5d24` subdirectory of the orb package. Then you simply run the example in the following way:

Example

```
gap> ReadPackage("orb","examples/co1F5d24/Co1OrbitOnPF5d24.g");
...
gap> ReadPackage("orb","examples/co1F5d24/Co1OrbitOnPF5d24.findall.g");
...
```

We will not go into the details of the first part of the computation here, as they are very similar to those reproduced in Section 11.1, and can be directly read off from the code in the file `Co1OrbitOnPF5d24.g`: We are using three helper subgroups $U_1 < U_2 < U_3 < Co_1$, where Co_1 has order $4\,157\,776\,806\,543\,360\,000 \approx 4.2 \cdot 10^{18}$, see [CCN+85] or the CTblLib package, and where $U_3 \cong 2_+^{1+8}.O_8(2)$ is the fifth maximal subgroup of Co_1, having order $89\,181\,388\,800 \approx 8.9 \cdot 10^{10}$, while $U_2 \cong [2^8]{:}S_6(2)$ is a maximal subgroup of U_3 of order $371\,589\,120 \approx 3.7 \cdot 10^8$, and $U_1 \cong 2^6{:}L_3(2)$ is a maximal subgroup of $S_6(2)$ of order $10\,752 \approx 1.1 \cdot 10^4$. The projective action comes from the irreducible 24-dimensional linear representation of the Schur cover $2.Co_1$ of Co_1, which by [Jan05] is the smallest faithful representation of $2.Co_1$ over the field GF(5), and the quotient spaces we use for the helper subgroups have dimensions 8, 8 and 16 respectively.

The details of the second part can be directly read off from the code in the file `Co1OrbitOnPF5d24.findall.g`:

Example

```
oo := InitOrbitBySuborbitList(setup,80);
l := MakeRandomLines(v,1000);
OrbitsFromSeedsToOrbitList(oo,l);
intervecs := CVEC_ReadMatFromFile(Filename(DirectoriesPackageLibrary("orb",""),
               "examples/co1F5d24/co1interestingvecs.cmat"));
OrbitsFromSeedsToOrbitList(oo,intervecs);
Length(oo!.obsos);
Sum(oo!.obsos,Size);
(5^24-1)/(5-1);
```

Note that this example needs about 2GB of main memory on a 32bit machine and probably nearly 4GB on a 64bit machine. However, the orbit lengths were previously unknown before they were computed with this program. The orbit lengths were needed to rule out a case in [Mal06].

11.4 The Baby Monster B acting on its 2A involutions

The example in this section shows how to enumerate the smallest non-trivial orbit of the Baby Monster group B. All the files for this example are located in the `examples/bmF2d4370` subdirectory of the orb package. You may simply run the example in the following way:

Example

```
gap> ReadPackage("orb","examples/bmF2d4370/BMOrbitOnF2d4370partI.g");
...
gap> ReadPackage("orb","examples/bmF2d4370/BMOrbitOnF2d4370partII.g");
...
```

In the sequel we comment in detail on how the necessary input data actually is prepared. We begin by loading the packages we are going to use.

Example

```
gap> LoadPackage("orb");
...
```

```
gap> LoadPackage("io");
...
gap> LoadPackage("cvec");
...
gap> LoadPackage("atlasrep");
...
gap> LoadPackage("chop");
...
gap> LoadPackage("genss");
...
```

The one-point stabilisers associated to the smallest non-trivial orbit of B are its largest maximal subgroups $E \cong 2.{}^2E_6(2).2$, which are the centralisers of its 2A involutions. Here E is a bicyclic extension of the twisted Lie type group ${}^2E_6(2)$, and has index $[B{:}E] = 13\,571\,955\,000 \approx 1.4 \cdot 10^{10}$, see [CCN+85] or the CTblLib package.

We first try to find a matrix representation of B such that the B-orbit we look for is realised as a set of vectors in the underlying vector space. The smallest faithful representation of B over the field GF(2), by [Jan05] having dimension 4370, springs to mind. Explicit matrices in terms of standard generators in the sense of [Wil96] are available in [Wil], and are accessibe through the atlasrep package. Moreover, we find generators of E by applying a straight line program, also available in the atlasrep package, expressing generators of E in terms of the generators of B.

Example

```
gap> gens := AtlasGenerators("B",1).generators;
[ <an immutable 4370x4370 matrix over GF2>,
  <an immutable 4370x4370 matrix over GF2> ]
gap> bgens := List(gens,CMat);
[ <cmat 4370x4370 over GF(2,1)>, <cmat 4370x4370 over GF(2,1)> ]
gap> slpbtoe := AtlasStraightLineProgram("B",1).program;;
gap> egens := ResultOfStraightLineProgram(slpbtoe,bgens);
[ <cmat 4370x4370 over GF(2,1)>, <cmat 4370x4370 over GF(2,1)> ]
```

We look for a non-zero vector being fixed by both generators of E. It turns out that the latter have a common fixed space of dimension 1. Then, since E is a maximal subgroup, the stabiliser in B of the non-zero vector v in that fixed space coincides with E.

Example

```
gap> x := egens[1]-egens[1]^0;;
gap> nsx := NullspaceMat(x);
<immutable cmat 2202x4370 over GF(2,1)>
gap> y := nsx * (egens[2]-egens[2]^0);;
gap> nsy := NullspaceMat(y);
<immutable cmat 1x2202 over GF(2,1)>
gap> v := nsy[1]*nsx;
<immutable cvec over GF(2,1) of length 4370>
```

Storing eight elements of GF(2) into 1 byte, to store a vector of length 4370 needs 547 bytes plus some organisational overhead resulting in about 580 bytes, hence to store the full B-orbit of v we need $580 \cdot [B{:}E] \approx 7.9 \cdot 10^{12}$ bytes. Hence we try to find helper subgroups suitable to achieve a saving factor of $\approx 10^4$, i. e. allowing to store only one out of $\approx 10^4$ vectors. To this end, we look for a pair $U_1 < U_2$ of helper subgroups such that $|U_2| \approx 10^5$, where we take into account that typically the overall saving factor achieved is somewhat smaller than the order of the largest helper subgroup.

By [CCN$^+$85], and a few computations with subgroup fusions using the CTblLib package, the derived subgroup $E' \cong 2.{}^2E_6(2)$ of E turns out to possess maximal subgroups $2 \times Fi_{22}$ and $2.Fi_{22}$, where Fi_{22} denotes one of the sporadic simple Fischer groups, and where the former constitute a unique conjugacy class with associated normalizers in E of shape $2 \times Fi_{22}.2$, while the latter consist of two conjugacy classes being self-normalising and interchanged by E.

Now Fi_{22} has a unique conjugacy class of maximal subgroups M_{12}, where the latter denotes one of the sporadic simple Mathieu groups; the subgroups M_{12} lift to a unique conjugacy class of subgroups M_{12} of $2.Fi_{22}$, which turn out to constitute a conjugacy class of subgroups of E different from the subgroups M_{12} being contained in Fi_{22}. Anyway, we have $|M_{12}| = 95\,040$, hence $U_2 = M_{12}$ seems to be a good candidate for the larger helper subgroup. In particular, there is a unique conjugacy class of maximal subgroups $L_2(11)$ of M_{12}, and since $|L_2(11)| = 660$ and $[M_{12}{:}L_2(11)] = 144$ letting $U_1 = L_2(11)$ seems to be a good candidate for the smaller helper subgroup. Recall that U_1 and U_2 are useful helper subgroups only if we are able to find suitable quotient modules allowing for the envisaged saving factor.

To find U_1 and U_2, we first try to find a subgroup Fi_{22} or $2.Fi_{22}$ of E. We start a random search, aiming at finding standard generators of either Fi_{22} or $2.Fi_{22}$, and we use `GeneratorsWithMemory` in order to be able to express the generators found as words in the generators of E. To accelerate computations we first construct a small representation of E; by [Jan05] the smallest faithful irreducible representation of Fi_{22} over GF(2) has dimension 78, hence we cannot do better for E; note that the latter is a representation of $\overline{E} := E/Z(E) \cong {}^2E_6(2).2$.

Example

```
gap> SetInfoLevel(InfoChop,2);
gap> m := Module(egens);
<module of dim. 4370 over GF(2)>
gap> r := Chop(m);
...
rec( ischoprecord := true,
  db := [ <abs. simple module of dim. 78 over GF(2)>,
          <trivial module of dim. 1 over GF(2)>,
          <abs. simple module of dim. 1702 over GF(2)>,
          <abs. simple module of dim. 572 over GF(2)> ],
  mult := [ 5, 4, 2, 1 ], acs := [ 1, 2, 3, 1, 4, 1, 1, 2, 2, 3, 1, 2 ],
  module := <reducible module of dim. 4370 over GF(2)> )
gap> i := Position(List(r.db,Dimension),78);;
gap> egens78 := GeneratorsWithMemory(RepresentingMatrices(r.db[i]));
[ <<immutable cmat 78x78 over GF(2,1)> with mem>,
  <<immutable cmat 78x78 over GF(2,1)> with mem> ]
```

By [Wil], standard generators a,b of Fi_{22} are given as follows: a is an element of the 2A conjugacy class of Fi_{22}, and b, ab, and $(ab)^4bab(abb)^2$ have order 13, 11, and 12, respectively; standard generators of $2.Fi_{22}$ are lifts of standard generators of Fi_{22} having the same order fingerprint. The 2A conjugacy class of Fi_{22} fuses into the 2A conjugacy class of $\overline{E}$, where the latter is obtained as the 11-th power of the unique conjugacy class of elements of order 22, and $\overline{E}$ has only one conjugacy class of elements of order 13.

Example

```
gap> o := Orb(egens78,StripMemory(egens78[1])^0,OnRight,rec(schreier := true,
>              lookingfor := function(o,x) return Order(x)=22; end));
<open orbit, 1 points with Schreier tree looking for sth.>
gap> Enumerate(o);
<open orbit, 393 points with Schreier tree looking for sth.>
```

```
gap> word := TraceSchreierTreeForward(o,PositionOfFound(o));
[ 1, 2, 1, 2, 2, 1, 2, 2, 1, 2, 2 ]
gap> g2a := Product(egens78{word})^11;
<<immutable cmat 78x78 over GF(2,1)> with mem>
gap> o := Orb(egens78,StripMemory(egens78[1])^0,OnRight,rec(schreier := true,
>             lookingfor := function(o,x) return Order(x)=13; end));
<open orbit, 1 points with Schreier tree looking for sth.>
gap> Enumerate(o);
<open orbit, 144 points with Schreier tree looking for sth.>
gap> word := TraceSchreierTreeForward(o,PositionOfFound(o));
[ 1, 2, 1, 2, 1, 2, 1, 2, 2 ]
gap> b := Product(egens78{word});
<<immutable cmat 78x78 over GF(2,1)> with mem>
```

We search through the $\overline{E}$-conjugates of g2a until we find a conjugate a together with b fulfilling the defining conditions of standard generators of Fi_{22}, and moreover fulfilling the relations of the associated presentation of Fi_{22} available in [Wil].

To find conjugates, we use the product replacement algorithm to produce pseudo random elements of $\overline{E}$. Assuming a genuine random search, the success probability of this approach is as follows: Letting $\overline{E'} := E'/Z(E') \cong {}^2E_6(2)$, out of the $|\overline{E'}|/|C_{\overline{E'}}(g2a)|$ conjugates of g2a there are $|C_{\overline{E'}}(b)|/|C_{\overline{E'}}(Fi_{22})| = |C_{\overline{E'}}(b)|$ elements together with the fixed element b giving standard generators of Fi_{22}. Since Fi_{22} has two conjugacy classes of elements of order 13, and there are three conjugacy classes of subgroups Fi_{22} of $\overline{E'}$, the success probability is $6 \cdot |C_{\overline{E'}}(g2a)| \cdot |C_{\overline{E'}}(b)|/|\overline{E'}| \approx 2 \cdot 10^{-5}$.

Example

```
gap> pr := ProductReplacer(egens78,rec(maxdepth := 150));
<product replacer nrgens=2 slots=12 scramble=100 maxdepth=150 steps=0 (rattle)>
gap> i := 0;;
gap> repeat
>      i := i + 1;
>      x := Next(pr);
>      a := g2a^x;
>    until IsOne((a*b)^11) and IsOne(((a*b)^4*b*a*b*(a*b*b)^2)^12) and
>      IsOne((a*b^2)^21) and IsOne(Comm(a,b)^3) and
>      IsOne(Comm(a,b^2)^3) and IsOne(Comm(a,b^3)^3) and
>      IsOne(Comm(a,b^4)^2) and IsOne(Comm(a,b^5)^3) and
>      IsOne(Comm(a,b*a*b^2)^3) and IsOne(Comm(a,b^-1*a*b^-2)^2) and
>      IsOne(Comm(a,b*a*b^5)^2) and IsOne(Comm(a,b^2*a*b^5)^2);
gap> i;
53271
```

Note that the initial state of the random number generator does influence this randomised result: it may very well be that you see some other value for i.

Due to a presentation being available we have proved that the elements found generate a subgroup Fi_{22}. If we had not had a presentation at hand, we might only have been able to find elements fulfilling the defining conditions of standard generators of Fi_{22}, but still generating a subgroup of another isomorphism type. In that case, for further checks we can use the following tools: We try to find a short orbit of vectors, and using a randomized Schreier-Sims algorithm gives a lower bound for the order of the group seen. However, we can use the action on the orbit to get a homomorphism into a permutation group, allowing to prove that the group generated indeed has Fi_{22} as a quotient.

Example

```
gap> S := StabilizerChain(Group(a,b),rec(TryShortOrbit := 30,
>                                         OrbitLengthLimit := 5000));
...
<stabchain size=64561751654400 orblen=3510 layer=1 SchreierDepth=8>
 <stabchain size=18393661440 orblen=2816 layer=2 SchreierDepth=7>
  <stabchain size=6531840 orblen=1680 layer=3 SchreierDepth=7>
   <stabchain size=3888 orblen=243 layer=4 SchreierDepth=5>
    <stabchain size=16 orblen=16 layer=5 SchreierDepth=2>
gap> Size(S)=Size(CharacterTable("Fi22"));
true
gap> p := Group(ActionOnOrbit(S!.orb,[a,b]));;
gap> DisplayCompositionSeries(p);
G (2 gens, size 64561751654400)
 | Fi(22)
1 (0 gens, size 1)
```

We now return to our original representation.

Example

```
gap> SetInfoLevel(InfoSLP,2);
gap> slpetofi22 := SLPOfElms([a,b]);
<straight line program>
gap> Length(LinesOfStraightLineProgram(slpetofi22));
278
gap> SlotUsagePattern(slpetofi22);;
gap> fgens := ResultOfStraightLineProgram(slpetofi22,egens);
...
[ <cmat 4370x4370 over GF(2,1)>, <cmat 4370x4370 over GF(2,1)> ]
gap> a := fgens[1];;
gap> b := fgens[2];;
gap> IsOne(b^13);
true
gap> IsOne((a*b)^11);
true
gap> IsOne((a*b^2)^21);
true
```

By construction the group generated by a,b is Fi_{22} or $2 \times Fi_{22}$ or $2.Fi_{22}$. Note that due to different seeds in the random number generator it is in fact possible at this stage that you have created a different group as displayed here! In our outcome, since a has even order, and both b and ab have odd order, we cannot possibly have $2 \times Fi_{22}$; and by the presentation of $2.Fi_{22}$ available in [Wil] the order of ab^2 being 21 implies that we cannot possibly have $2.Fi_{22}$ either. Hence we indeed have found standard generators of Fi_{22}. If we had hit one of the cases $2 \times Fi_{22}$ or $2.Fi_{22}$, we could just continue the above search until we find a subgroup Fi_{22}, or using the above order fingerprint we could easily modify the elements found to obtain standard generators of either Fi_{22} or $2.Fi_{22}$.

Now, standard generators of $U_2 = M_{12}$ in terms of standard generators of Fi_{22}, and generators of $U_1 = L_2(11)$ in terms of standard generators of M_{12} are accessible in the atlasrep package. Note that if we had found a subgroup $2.Fi_{22}$ above, since M_{12} lifts to a subgroup $2 \times M_{12}$ of $2.Fi_{22}$, it would again be easy to find standard generators of M_{12} from the generators of M_{12} or $2 \times M_{12}$ respectively provided by the atlasrep package. Anyway, the next task is to find good quotient modules such that the helper subgroups have longish orbits on vectors. To this end, we restrict to M_{12} and compute the radical series of the restricted module.

Example

```
gap> slpfi22tom12 := AtlasStraightLineProgram("Fi22",14).program;;
gap> slpm12tol211 := AtlasStraightLineProgram("M12",5).program;;
gap> mgens := ResultOfStraightLineProgram(slpfi22tom12,fgens);
[ <cmat 4370x4370 over GF(2,1)>, <cmat 4370x4370 over GF(2,1)> ]
gap> lgens := ResultOfStraightLineProgram(slpm12tol211,mgens);
[ <cmat 4370x4370 over GF(2,1)>, <cmat 4370x4370 over GF(2,1)> ]
gap> m := Module(mgens);;
gap> r := Chop(m);;
...
gap> rad := RadicalSeries(m,r.db);
...
rec(
  db := [ <abs. simple module of dim. 144 over GF(2)>,
          <abs. simple module of dim. 44 over GF(2)>,
          <simple module of dim. 32 over GF(2) splitting field degree 2>,
          <abs. simple module of dim. 10 over GF(2)>,
          <trivial module of dim. 1 over GF(2)> ],
  module := <reducible module of dim. 4370 over GF(2)>,
  basis := <immutable cmat 4370x4370 over GF(2,1)>,
  ibasis := <immutable cmat 4370x4370 over GF(2,1)>,
  cfposs := [ [ [ 1 .. 144 ], [ 145 .. 288 ], [ 289 .. 432 ], [ 433 .. 576 ],
...
  isotypes := [ [ 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3,
                  3, 3, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ],
...
  isradicalrecord := true )
```

We observe that there are faithful irreducible quotients of dimensions 10, 32, 44, and 144. Since we look for a quotient module such that M_{12} has many regular orbits on vectors, we ignore the irreducible module of dimension 10. We consider the one of dimension 32.

Example

```
gap> i := Position(List(rad.db,Dimension),32);;
gap> mgens32 := RepresentingMatrices(rad.db[i]);
[ <immutable cmat 32x32 over GF(2,1)>, <immutable cmat 32x32 over GF(2,1)> ]
gap> OrbitStatisticOnVectorSpace(mgens32,95040,30);
Found length     95040, have now   24 orbits, average length: 93060
```

This is excellent indeed. Hence we pick a summand of dimension 32 in the first radical layer, and apply the associated base change to all the generators.

Example

```
gap> bgens := List(bgens,x->rad.basis*x*rad.ibasis);;
gap> egens := List(egens,x->rad.basis*x*rad.ibasis);;
gap> fgens := List(fgens,x->rad.basis*x*rad.ibasis);;
gap> mgens := List(mgens,x->rad.basis*x*rad.ibasis);;
gap> lgens := List(lgens,x->rad.basis*x*rad.ibasis);;
gap> j := Position(rad.isotypes[1],i);;
gap> l := rad.cfposs[1][j];;
gap> Append(l,Difference([1..4370],l));
gap> bgens := List(bgens,x->ORB_PermuteBasisVectors(x,l));;
gap> egens := List(egens,x->ORB_PermuteBasisVectors(x,l));;
gap> fgens := List(fgens,x->ORB_PermuteBasisVectors(x,l));;
gap> mgens := List(mgens,x->ORB_PermuteBasisVectors(x,l));;
gap> lgens := List(lgens,x->ORB_PermuteBasisVectors(x,l));;
```

We consider the irreducible quotient module of M_{12} of dimension 32, whose restriction to $L_2(11)$ turns out to be is semisimple. The irreducible quotients of dimension 10 are too small to have too many regular orbits, but the direct sum of two of them turns out to work fine.

Example

```
gap> lgens32 := List(lgens,x->ExtractSubMatrix(x,[1..32],[1..32]));
[ <cmat 32x32 over GF(2,1)>, <cmat 32x32 over GF(2,1)> ]
gap> m := Module(lgens32);;
gap> r := Chop(m);
...
gap> soc := SocleSeries(m,r.db);
...
rec( issoclerecord := true,
  db := [ <simple module of dim. 10 over GF(2) splitting field degree 2>,
          <trivial module of dim. 1 over GF(2)>,
          <abs. simple module of dim. 10 over GF(2)> ],
  module := <reducible module of dim. 32 over GF(2)>,
  basis := <cmat 32x32 over GF(2,1)>, ibasis := <cmat 32x32 over GF(2,1)>,
  cfposs := [ [ [ 1 .. 10 ], [ 11 ], [ 12 ], [ 13 .. 22 ], [ 23 .. 32 ] ] ],
  isotypes := [ [ 1, 2, 2, 3, 3 ] ] )
gap> i := Position(List(soc.db,x->[Dimension(x),DegreeOfSplittingField(x)]),
>                               [10,1]);;
gap> j := Position(soc.isotypes[1],i);;
gap> l := Concatenation(soc.cfposs[1]{[j,j+1]});;
gap> lgens32 := List(lgens32,x->soc.basis*x*soc.ibasis);
[ <cmat 32x32 over GF(2,1)>, <cmat 32x32 over GF(2,1)> ]
gap> lgens20 := List(lgens32,x->ExtractSubMatrix(x,l,l));
[ <cmat 20x20 over GF(2,1)>, <cmat 20x20 over GF(2,1)> ]
gap> OrbitStatisticOnVectorSpace(lgens20,660,30);
Found length       660, have now 4401 orbits, average length: 598
```

We apply the appropriate base change to all the generators.

Example

```
gap> t := ORB_EmbedBaseChangeTopLeft(soc.basis,4370);
<cmat 4370x4370 over GF(2,1)>
gap> ti := ORB_EmbedBaseChangeTopLeft(soc.ibasis,4370);
<cmat 4370x4370 over GF(2,1)>
gap> bgens := List(bgens,x->t*x*ti);;
gap> egens := List(egens,x->t*x*ti);;
gap> fgens := List(fgens,x->t*x*ti);;
gap> mgens := List(mgens,x->t*x*ti);;
gap> lgens := List(lgens,x->t*x*ti);;
gap> Append(l,Difference([1..4370],l));
gap> bgens := List(bgens,x->ORB_PermuteBasisVectors(x,l));;
gap> egens := List(egens,x->ORB_PermuteBasisVectors(x,l));;
gap> fgens := List(fgens,x->ORB_PermuteBasisVectors(x,l));;
gap> mgens := List(mgens,x->ORB_PermuteBasisVectors(x,l));;
gap> lgens := List(lgens,x->ORB_PermuteBasisVectors(x,l));;
```

Having reached the ultimate choice of basis, we recreate the fixed vector v.

Example

```
gap> x := egens[1]-egens[1]^0;;
gap> nsx := NullspaceMat(x);;
gap> y := nsx * (egens[2]-egens[2]^0);;
gap> nsy := NullspaceMat(y);;
gap> v := nsy[1]*nsx;;
```

Finally we need small faithful permutation representations of the helper subgroups.

Example

```
gap> mgens32 := List(mgens,x->ExtractSubMatrix(x,[1..32],[1..32]));
[ <cmat 32x32 over GF(2,1)>, <cmat 32x32 over GF(2,1)> ]
gap> S := StabilizerChain(Group(mgens32),rec(TryShortOrbit := 10));
...
<stabchain size=95040 orblen=3960 layer=1 SchreierDepth=7>
 <stabchain size=24 orblen=24 layer=2 SchreierDepth=4>
gap> p := Group(ActionOnOrbit(S!.orb,mgens32));
<permutation group with 2 generators>
gap> i := SmallerDegreePermutationRepresentation(p);;
gap> pp := Group(List(GeneratorsOfGroup(p),x->ImageElm(i,x)));
<permutation group with 2 generators>
gap> m12 := MathieuGroup(12);;
gap> i := IsomorphismGroups(pp,m12);;
gap> mpermgens := List(GeneratorsOfGroup(pp),x->ImageElm(i,x));
[ (5,7)(6,11)(8,9)(10,12), (1,10,3)(2,11,12)(4,5,6)(7,9,8) ]
gap> lpermgens := ResultOfStraightLineProgram(slpm12tol211,mpermgens);
[ (1,8)(2,5)(3,9)(4,7)(6,11)(10,12), (1,8,3)(2,7,12)(4,6,9)(5,11,10) ]
```

We could just go on from here, however, sometimes it is useful to save all the created data to disk.

Example

```
gap> f := IO_File("data.gp","w");;
gap> IO_Pickle(f,"seed");;
gap> IO_Pickle(f,v);;
gap> IO_Pickle(f,"generators");;
gap> IO_Pickle(f,bgens);;
gap> IO_Pickle(f,egens);;
gap> IO_Pickle(f,fgens);;
gap> IO_Pickle(f,mgens);;
gap> IO_Pickle(f,lgens);;
gap> IO_Pickle(f,"permutations");;
gap> IO_Pickle(f,mpermgens);;
gap> IO_Pickle(f,lpermgens);;
gap> IO_Close(f);;
```

This can be loaded again, in particular into a new GAP session, as follows.

Example

```
gap> LoadPackage("orb");;
...
gap> LoadPackage("cvec");;
...
gap> f := IO_File("data.gp");
<file fd=4 rbufsize=65536 rpos=1 rdata=0>
```

```
gap> IO_Unpickle(f);
"seed"
gap> v:=IO_Unpickle(f);;
gap> IO_Unpickle(f);
"generators"
gap> bgens := IO_Unpickle(f);;
gap> egens := IO_Unpickle(f);;
gap> fgens := IO_Unpickle(f);;
gap> mgens := IO_Unpickle(f);;
gap> lgens := IO_Unpickle(f);;
gap> IO_Unpickle(f);
"permutations"
gap> mpermgens := IO_Unpickle(f);;
gap> lpermgens := IO_Unpickle(f);;
gap> IO_Close(f);;
```

Now we are prepared to actually run the orbit enumeration. Note that for the following memory estimates we assume that we are running things on a 64bit machine. On a 32bit machine the overhead is smaller. We expect that all the vectors in the smaller quotient of dimension 20 will enumerated; needing 3 bytes per vector for the actual data which results in 40 bytes including overhead, this amounts to $40 \cdot 2^{20} \approx 42$ MB of memory space. Since $2^{32} \approx 4.3 \cdot 10^9$ is less than $[B:E]$, we also expect that the larger quotient of dimension 32 will be enumerated completely, by $L_2(11)$-orbits; needing 4 bytes per vector for the actual data resulting in 40 bytes including overhead, and assuming a saving factor as suggested by `OrbitStatisticOnVectorSpace` yields an estimated memory requirement of $40 \cdot 2^{32} \cdot 1/598 \approx 287$ MB. For the large B-orbit, being enumerated by M_{12}-orbits, we similarly get an estimated memory requirement of $584 \cdot [B:E] \cdot 1/93060 \approx 85$ MB.

Example

```
gap> setup := OrbitBySuborbitBootstrapForVectors(
>             [lgens,mgens,bgens],[lpermgens,mpermgens,[(),()]],
>             [660,95040,4154781481226426191177580544000000],[20,32],rec());
#I  Calculating stabilizer chain for whole group...
#I  Trying smaller degree permutation representation for U2...
#I  Trying smaller degree permutation representation for U1...
#I  Enumerating permutation base images of U_1...
#I  Looking for U1-coset-recognising U2-orbit in factor space...
#I  OrbitBySuborbit found 100% of a U2-orbit of size 95 040
#I  Found 144 suborbits (need 144)
<setup for an orbit-by-suborbit enumeration, k=2>
gap> o := OrbitBySuborbitKnownSize(setup,v,3,3,2,51,13571955000);
#I  OrbitBySuborbit found 100% of a U2-orbit of size 1
#I  OrbitBySuborbit found 100% of a U2-orbit of size 23 760
...
#I  OrbitBySuborbit found 51% of a U3-orbit of size 13 571 955 000
<orbit-by-suborbit size=13571955000 stabsize=306129918735099415756800 (
51%) saving factor=56404>
```

Indeed the saving factor actually achieved is smaller than the best possible estimate given above, but it still has the same order of magnitude.

References

[AVM62] G. Adelson-Velskii and Landis E. M. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962. Russian. 38

[CCN$^+$85] J[ohn] H. Conway, R[obert] T. Curtis, S[imon] P. Norton, R[ichard] A. Parker, and R[obert] A. Wilson. *Atlas of finite groups*. Oxford University Press, 1985. 57, 58, 59, 60, 61

[HNN06] Gerhard Hiss, Max Neunhöffer, and Felix Noeske. The 2-modular characters of the Fischer group Fi_{23}. *J. Algebra*, 300(2):555–570, 2006. 58

[Jan05] C. Jansen. The minimal degrees of faithful representations of the sporadic simple groups and their covering groups. *LMS J. Comput. Math.*, 8:122–144, 2005. 59, 60, 61

[Knu97] Donald Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, third edition, 1997. 38, 43

[Mal06] Gunter Malle. Fast-einfache Gruppen mit langen Bahnen in absolut irreduzibler Operation. *J. Algebra*, 300(2):655–672, 2006. 56, 59

[MNW07] J. Müller, M. Neunhöffer, and R. A. Wilson. Enumerating big orbits and an application: B acting on the cosets of Fi_{23}. *J. Algebra*, 314(1):75–96, 2007. 45, 56

[Mül08] J. Müller. On the action of the sporadic simple baby monster group on its conjugacy class $2B$. *LMS J. Comput. Math.*, 11:15–27, 2008. 56

[Wil] Robert A. Wilson. ATLAS of Finite Group Representations. http://brauer.maths.qmul.ac.uk/Atlas. 60, 61, 62, 63

[Wil96] R[obert] A. Wilson. Standard generators for sporadic simple groups. *Journal of Algebra*, 184:505–515, 1996. 60

Index