

Vole

Vole organises lengthy explorations: Backtrack search in permutation groups with graphs 0.6.0

9 January 2025

Mun See Chang
Christopher Jefferson
Wilf A. Wilson

Mun See Chang Email: msc2@st-andrews.ac.uk
Address: School of Computer Science, University of St Andrews,
St Andrews, Fife, KY16 9SX, Scotland

Christopher Jefferson Email: caj21@st-andrews.ac.uk
Homepage: <https://heather.cafe>
Address: School of Computer Science, University of St Andrews,
St Andrews, Fife, KY16 9SX, Scotland

Wilf A. Wilson Email: gap@wilf-wilson.net
Homepage: <https://wilf.me>
Address: School of Computer Science, University of St Andrews,
St Andrews, Fife, KY16 9SX, Scotland

Abstract

VOLE is a GAP package that implements algorithms in finite permutation groups, such as canonical images, stabilisers, and subgroup intersection. These are all implemented as part of the general 'graph backtracking' framework.

Copyright

© 2025 by Christopher Jefferson, Mun See Chang, and Wilf A. Wilson.
Vole is licensed under the Mozilla Public License, version 2.0.

Acknowledgements

The authors would like to thank the [Royal Society](#) (grant codes RGF\EA\181005 and URF\R\180015) for their financial support at the time that Vole was created.

Contents

1	Introduction to Vole	5
1.1	High level overview of Vole and this manual	5
1.2	Vole’s relation to GAP and its other packages	6
1.3	Work in progress	7
1.4	Performance	7
2	Setting up Vole	9
2.1	Dependencies	9
2.2	Installing Vole	10
2.3	Loading Vole	11
2.4	Running Vole’s test suite	11
2.5	Compiling the documentation for Vole	12
3	Tutorial	13
3.1	Simple interface	13
3.2	The full Vole interface	15
4	Emulating traditional interfaces with Vole	18
4.1	The concept	18
4.2	The Vole record	19
4.3	The group actions built into Vole	19
4.4	Vole functions emulating built-in GAP functions	20
4.5	Vole functions emulating the images package	24
4.6	Vole functions emulating the Digraphs package	26
5	The native Vole interface	29
5.1	The VoleFind record	29
5.2	Searching for groups, cosets, and representatives with the native interface	29
5.3	Canonising with the native Vole interface	32
6	Constraints	36
6.1	Bounds associated with a constraint or refiner	36
6.2	The concept of constraints	37
6.3	The Constraints record	37
6.4	Constraints via the Constraint record	38

7	Refiners in Vole	44
7.1	Refiners	44
7.2	The VoleRefiner record	44
7.3	Vole refiners via the VoleRefiner record	44
7.4	Choosing a refiner for a given constraint	46
8	Expert use of Vole	47
8.1	InfoVole – getting extra information about a Vole search	47
8.2	Options with the native Vole interface	47
	References	50
	Index	51

Chapter 1

Introduction to Vole

Vole is a GAP [GAP21] package. The purpose of this manual is to describe the capabilities of Vole, and to give directions, hints, and explanations about its use.

Vole is a very general framework. The easiest way to learn about the capabilities of Vole is by reading the tutorial (Chapter 3), to see a number of worked examples.

In this manual, a *permutation* is defined on the set of all positive integers, but moves only finitely many points. Unless otherwise stated, a *group* is a finite permutation group (which, by the definition of a permutation, necessarily has a finite set of moved points); and a *coset* is a left or right coset of such a group determined by some permutation.

In full generality, Vole can be used to find permutations (groups given by strong generating sets, cosets thereof, or individual representative elements) that satisfy conjunctions of particular kinds of properties, or *constraints*; details of these properties are given later. Problems that can be formulated in this way include, for example, the computation of normalisers, set stabilisers, subgroup intersections, and graph automorphisms, and testing subgroup conjugacy and graph isomorphism.

Vole can also compute canonical labellings and images (also known as ‘canonical forms’) of various kinds of objects under an action of an arbitrary permutation group. This includes, for example, the canonical image of a digraph under the natural action of an arbitrary permutation group, or the canonical image of a permutation group under conjugation by another. *As far as we are aware, most of this canonical image functionality is not possible with existing tools.*

In order to offer all of this functionality, Vole implements the *graph backtracking* framework, which was introduced in [JPWW21], and which generalises the *partition backtracking* framework of Jeffrey Leon [Leo91], [Leo97]. The functionality for canonical labellings and canonical images is underpinned, in addition, by work that is in preparation for publication, and which builds on [JJPW19].

While the GAP interface of this package is written in GAP, the main algorithms are written in Rust, for reasons of fun and performance.

1.1 High level overview of Vole and this manual

Vole aims to provide a high-performance implementation of the graph backtracking framework of [JPWW21], [JPWW19].

Graph backtracking is a form of depth-first backtrack search that can be performed in a finite symmetric group, and which uses labelled digraphs to represent various aspects of the problem at hand in order to guide the search and prune the search space. The specification of the problem is given

by a collection of *constraints*, and a collection of *refiners* are used to prune the search according to those constraints.

The rest of this introduction discusses the place of **Vole** in the GAP ‘ecosystem’, and gives information about its ongoing development. Installation instructions for **Vole** are given in Chapter 2, and Chapter 3 gives a tutorial for the package. Chapter 4 describes an interface to **Vole** that emulates the existing related functionality of GAP and its packages. Chapter 5 documents the main interface to **Vole**. Chapter 6 describes the constraints that can be used with **Vole**, while Chapter 7 explains refiners, and gives details of those that are available. Chapter 8 gives lower-level details, and instructions for the expert use of **Vole**.

If you encounter problems, or have any other feedback for the authors, then please [create an issue on our issue tracker](#), or contact us by any other appropriate means. See the title page??? of this manual for our contact details. We published **Vole** so that our ideas and algorithms could be used by, and be useful to, others in the community. But without feedback, it can be difficult to know whether we are achieving this aim. Therefore we are keen to hear your comments, remarks, and complaints.

1.2 **Vole’s relation to GAP and its other packages**

The functionality of **Vole** overlaps with that of GAP itself, and some of its packages. However, **Vole’s** functionality goes further than all of them, and has the potential to offer better performance in many cases where there is an overlap. The performance of **Vole** is discussed in Section 1.4.

GAP itself

GAP [GAP21] contains an implementation of Jeffrey Leon’s partition backtrack framework [Leo91], [Leo97] at the GAP level, as improved upon by Heiko Theißen [The97]. Therefore GAP already offers some of the functionality that **Vole** provides; the overlap is explored further in Section 4.4.

ferret

Christopher Jefferson’s **ferret** package [Jef21] for GAP provides a C++ reimplementaion of partition backtracking, which contains the orbital graph developments of [JPW19]. **Vole** should be seen as the spiritual successor to **ferret**.

images

The **images** package [JJPW21] provides a GAP-level implementation of the custom breadth-first search that was introduced in [JJPW19] to compute the *canonical image* of any set of points under the action of an arbitrary permutation group. This algorithm built upon the original algorithm of Steve Linton [Lin04] for computing *minimal images* of sets of points, which is available in the GRAPE package [Soi21] in `lib/smallestimage.g`, as `SmallestImageSet`. The overlap of **Vole’s** functionality with **images** is detailed in Section 4.5.

Digraphs and GRAPE

The **Digraphs** package [BJM⁺21] offers interfaces to **nauty** [MP14] (via the `NautyTracesInterface`) and **bliss** [JK07] for computing automorphism groups and canonical images of graphs and digraphs, and their labelled variants. The overlap of **Vole’s** functionality with **Digraphs** is detailed in Section 4.6. GRAPE [Soi21] also provides an interface to **nauty** for computing automorphism groups of graphs, and computing canonical images of graphs in the symmetric group.

BacktrackKit and GraphBacktracking

BacktrackKit [JW21a] and **GraphBacktracking** [JW21b] contain GAP-level reference implementations of partition backtracking and graph backtracking, respectively. They are intended to

demonstrate readable code implementing these frameworks, but they are not intended to have high performance, and they are deliberately not optimized.

1.3 Work in progress

Vole is a work in progress – the code and its underlying mathematical theory are being developed concurrently. Therefore, some aspects of Vole are liable to change in future versions, although we will aim to maintain the primary interfaces as described in Chapters 4, 5, and 6. However, some of the lower-level details described in Sections 7 and 8 are more likely change.

The main focus of our future work on Vole will be on improving the performance of Vole, through both technical and mathematical means. This may involve changes to the design of the algorithm itself, and the improvement or addition of refiners, especially for subgroup normalisers and conjugacy. See Section 1.4 for further comments on the performance of Vole.

1.4 Performance

Our aims are for Vole to have good performance in general, and to have relatively high performance for some classes of problems that existing tools tend to find very difficult. As mentioned in Section 1.3, we are still working towards these aims.

If you encounter disappointing and/or surprising performance for your use-case, please let us know, by [creating an issue on our issue tracker](#), or contacting us by any other appropriate means. See the title page of this manual for our contact details. This will help us to improve Vole.

The experiments in [JPWW21] showed that for some classes of problems, graph backtracking typically performs a far smaller search than does partition backtracking. Although this experimental data is very encouraging, it is clearly inherently more expensive to compute with graphs than it is to compute with ordered partitions. This means that the time gained by having a smaller number of nodes in a search does not necessarily immediately outweigh the time required to perform a larger amount of computation at each node. Therefore, in rough terms, the focus of future work is on reducing the number of search nodes further (with more sophisticated refiners), and reducing the amount of work required at each node of search (with technical optimisations).

Where there is overlap between Vole and existing functionality in the GAP ecosystem, as described in Section 1.2, we aim for Vole to perform reasonably competitively.

However, roughly speaking, there are certain problems that can be represented by ordered partitions just as well as they can by graphs. In these cases, it follows that there is nothing to be gained by using graph backtracking, while the additional overhead required by graph backtracking may make it slower overall. Vole does not yet attempt to mitigate this issue. For example, we do not yet expect Vole to match the performance of nauty (available through the Digraphs and GRAPE packages) at computing automorphisms of graphs, and canonical images of graphs in a symmetric group.

In addition, we do not expect Vole to beat images at computing canonical images of sets in groups.

There are further reasons why existing implementations may beat the current version of Vole, in some situations.

- *Very few special cases:* Vole passes almost every problem that it receives on to the full graph backtracking algorithm, with its corresponding overheads and poor worst-case complexity. On

the other hand, other implementations (particularly those in the **GAP** library) may provide multiple competing methods for solving a problem. Some of these methods might only apply in special cases (such as for simple groups, or abelian group, or primitive groups), and some of them might not even involve search at all. From these different methods, the system can use heuristics to choose the most appropriate one to run in each instance.

- *First-time compilation of Vole:* When the first **Vole** search is executed in any particular **GAP** session, **Vole** tests whether its Rust component is compiled, and it builds the component if necessary. The check itself can take a noticeable amount of time, but compilation takes much longer. To minimise the effect, we recommend that you manually recompile **Vole** when you update your **Vole** installation via the `make` command, as described in Section 2.2.
- *Communication between the GAP layer and the Rust component:* There is also a delay that is incurred when the **GAP**-level interface of **Vole** communicates with the Rust component of **Vole**. This is exacerbated by the fact that **Vole** does not yet include its own Rust-level implementation of stabiliser chains, and so it uses **GAP**'s stabiliser chains instead. This means that **GAP** and the Rust component must communicate heavily during a typical search, and this can incur a significant time penalty. Implementing Rust-level stabiliser chains is a work in progress.
- *Fortune:* The nature of backtrack search means that one algorithm may beat another just by being 'lucky', and (by chance) stumbling upon particularly fruitful branches early in search. Conversely, an algorithm may be 'unlucky' and fall down a particular unfruitful branch that a different algorithm accidentally avoids.

These factors together highlight the principle that, for users for whom performance is very important, it makes sense to experiment with the range of available tools when running computations.

Please also make sure that you are running the newest version of **Vole**, since newer versions should have better performance.

Chapter 2

Setting up Vole

This chapter contains instructions for getting **Vole** up and running.

If you encounter problems, or have any other feedback for the authors, then please [create an issue on our issue tracker](#), or contact us by any other appropriate means. See the title page??? of this manual for our contact details. We published **Vole** so that our ideas and algorithms could be used by, and be useful to, others in the community. But without feedback, it can be difficult to know whether we are achieving this aim. Therefore we are keen to hear your comments, remarks, and complaints.

2.1 Dependencies

Vole requires GAP 4.13.0 or newer and various other GAP packages to run, and Rust version 1.48 or newer is required to compile **Vole**.

Please note that if you are using **Vole** via a GAP Cygwin release (i.e. if you installed GAP in Windows via an installer that included **Vole**), then **Vole** came ready-compiled, and the included versions of GAP and its packages are sufficient to run **Vole** out-of-the-box. Therefore you may skip straight to Section 2.3.

In order to compile **Vole**, Rust must be installed and available with the `cargo` command, in version at least 1.48. You can test for Rust by running `cargo --version` in the terminal. You should obtain a version number in response, such as:

<pre>\$ cargo --version cargo 1.57.0</pre>	Example
--	---------

If Rust is not available, then running the command `cargo --version` will give an error such as `cargo: command not found`. If this is the case, then please follow the instructions at <https://www.rust-lang.org/tools/install> or <https://rustup.rs> for installing Rust.

In addition, the following GAP packages must be available in order to use **Vole**. This means that **Vole** will not load if one or more of these packages cannot be loaded in your GAP installation in a new enough version.

- [Digraphs](#) version 1.1.1, or newer [BJM⁺21].
- [IO](#) version 4.7.0, or newer [Neu21].
- [datastructures](#) version 0.2.6, or newer [JHPL21].

- [json](#) version 2.0.1, or newer [[Jef20](#)].

These packages, and GAP itself, in turn have their own dependencies, and so Vole inherits these additional dependencies. If one of the above packages does not load in your GAP installation, then please consult the documentation for that package in order to resolve the problem.

The following GAP packages are necessary to use a small, non-core subset of Vole's features, and are therefore optional.

- [OrbitalGraphs](#) version 0.1, or newer (required for `Vole.TwoClosure` ([4.4.7](#))).

The following GAP packages are needed to run Vole's test suite, but they are not required to use the rest of Vole, and are therefore also optional.

- [QuickCheck](#) version 0.1, or newer.
- [ferret](#) version 1.0.2, or newer [[Jef21](#)].

See Section [2.4](#) for more information about running the tests included in Vole.

In order to compile the Vole documentation, [AutoDoc](#) version 2019.09.04 or newer must be available. This step is usually unnecessary, and is not required in order to use Vole. See Section [2.5](#) for more information about compiling the documentation.

2.2 Installing Vole

Please note that if you are using Vole via a GAP Cygwin release (i.e. if you installed GAP in Windows via an installer that included Vole), then Vole came ready-compiled, and the included versions of GAP and its packages are sufficient to run Vole out-of-the-box. Therefore you may skip straight to Section [2.3](#).

Otherwise, if you do not yet have a copy of Vole, you should download the latest version of Vole.

If you have the [PackageManager](#) package installed, then you should be able to use it to download and compile Vole. Please consult [the PackageManager documentation](#) for full instructions of how to do this, but the commands should look something like:

Example

```
gap> # Warning: this won't work until PackageManager has been taught about Vole!
gap> LoadPackage("PackageManager");
gap> InstallPackage("vole");
gap> CompilePackage("vole");
```

If this succeeds, you may skip to Section [2.3](#).

Alternatively, you can manually download the latest `.tar.gz` Vole archive from [the Vole website](#) and extract the archive into your GAP installation's `pkg` subdirectory. The commands to achieve this may look something like:

Example

```
$ cd ~/gap/pkg
$ curl -O https://github.com/peal/vole/releases/download/vX.Y.Z/vole-X.Y.Z.tar.gz
$ tar xf vole-X.Y.Z.tar.gz
```

In order to compile Vole, change to your Vole directory and run `make`. This should look something like:

Example

```
$ cd ~/gap/pkg/vole-X.Y.Z
$ make
cd rust && cargo build --bins && cargo build --release --bins
[...]
    potentially copious output omitted here
    ...]

Compiling rust-vole v0.1.1
Finished dev [unoptimized + debuginfo] target(s) in 8.93s
Compiling rust-vole v0.1.1
Finished release [optimized + debuginfo] target(s) in 30.06s
```

Vole should now be ready to load. See Section 2.3.

2.3 Loading Vole

If Vole is available to load, then it can be loaded by calling the function `LoadPackage` (**Reference: LoadPackage**) with the argument "vole", from within the relevant GAP installation. This should return true.

Example

```
gap> LoadPackage("vole", false);
true
```

If `LoadPackage` instead returns fail, then Vole has failed to load. This is probably because GAP cannot find Vole or Vole is not compiled (Section 2.2), or one of the dependencies of Vole is not available (Section 2.1). If Vole fails to load, then please see the `LoadPackage` (**Reference: LoadPackage**) documentation for ways to help diagnose the problem further.

If you encounter problems, or have any other feedback for the authors, then please [create an issue on our issue tracker](#), or contact us by any other appropriate means. See the title page??? of this manual for our contact details. We published Vole so that our ideas and algorithms could be used by, and be useful to, others in the community. But without feedback, it can be difficult to know whether we are achieving this aim. Therefore we are keen to hear your comments, remarks, and complaints.

Otherwise, Vole is ready to use. To reassure yourself, you can quickly test that Vole is behaving as intended by quickly running a search, such that the following one:

Example

```
gap> A4 := AlternatingGroup(4);
Alt( [ 1 .. 4 ] )
gap> D8 := DihedralGroup(IsPermGroup, 8);
Group([ (1,2,3,4), (2,4) ])
gap> Vole.Intersection(A4, D8) = Group([ (1,2)(3,4), (1,4)(2,3) ]);
true
```

If you wish to test your Vole installation more thoroughly, please see Section 2.4.

2.4 Running Vole's test suite

Optionally, if you wish to thoroughly test your Vole installation, then you can run the test suite. As well the dependencies of Vole itself, the test suite requires the GAP packages [ferret](#) (version 1.0.2 or newer) and [QuickCheck](#) (version 0.1 or newer) to be available.

There are two files that can be invoked to run extensive tests of **Vole**: `run-tests.sh` and `tst/testall.g`.

The `run-tests.sh` script, which is located in the root directory of **Vole**, begins by testing the Rust component directly, and then it tests the **GAP**-level interface to the Rust component. This script requires that the command `gap`, in your terminal environment, is set up to load the appropriate version of **GAP** for **Vole**.

Example

```
$ ./run-tests.sh
```

To run only the **GAP**-level tests, please read the file `testall.g` of **Vole**'s `tst` directory into **GAP**.

Example

```
gap> Read("tst/testall.g");
```

2.5 Compiling the documentation for **Vole**

Released versions of **Vole** are distributed with precompiled documentation, so compiling the documentation should be an unnecessary for most users. However, development versions of **Vole** do not include precompiled documentation.

To compile the manual, it suffices to read **Vole**'s `makedoc.g` file into **GAP**.

This can be done directly within **GAP** via the function `Read` (**Reference: Read**):

Example

```
gap> Read("makedoc.g");
```

Alternatively, the `makedoc.g` file can be read into **GAP** by giving its path as an argument to **GAP**.

Example

```
$ gap makedoc.g
```

Finally, running `make doc` in the **Vole** directory should compile the documentation of **Vole** if **Vole** is installed in the `pkg` subdirectory of the appropriate **GAP** installation.

Example

```
$ make doc
```

Chapter 3

Tutorial

There are two interfaces to Vole, the 'Simple Interface', which are methods which start with `Vole.`, and the 'Full Interface', which are methods which start with `VoleFind.`. The simple interface is no less efficient, each of the simple interface methods just call a problem in the full interface. Some problems cannot be expressed in the simple interface, in particular problems which involve solving multiple problems simultaneously.

3.1 Simple interface

The 'simple interface' to Vole consists of functions whose names begin `Vole.`. These methods implement existing methods in core GAP, and some GAP packages.

These functions are given as follows (grouped into sections:)

The following functions have very high quality implementations. The `Canonical` methods are described below, and are otherwise implemented in the `Images` package.

- `CanonicalImage`
- `CanonicalPerm`
- `CanonicalImagePerm`
- `RepresentativeAction`
- `IsConjugate`
- `Stabilizer`

These functions are implemented in `Digraphs`. They are generally slower than the functions provided there, but can be faster for multi-edge and edge-coloured graphs. Vole is much better when we want to search on a group other than the automorphism group on the vertices.

- `AutomorphismGroup`
- `CanonicalDigraph`
- `DigraphCanonicalLabelling`

- `IsIsomorphicDigraph`
- `IsomorphismDigraphs`
- These functions are standard group algorithms. Vole can sometimes outperform GAP, particularly when cosets are involved (for example coset intersection), but GAP is often faster for many problems.
- `TwoClosure`
- `Centralizer`
- `Normalizer`
- `Intersection`
- `Centraliser`

3.1.1 Simple computation of canonical images

Finding a canonical image always requires two arguments:

- The group in which we are searching.
- The object whose canonical image we want to find.

As a simple example, let's consider 3 sets, and if they are in the same orbit of M_{12} .

Example

```
gap> LoadPackage("vole", false);;
gap> s := [1,2,3,4,5,6];
[ 1, 2, 3, 4, 5, 6 ]
gap> t := [2,4,6,8,10,12];
[ 2, 4, 6, 8, 10, 12 ]
gap> u := [3,6,9,12,15,18];
[ 3, 6, 9, 12, 15, 18 ]
gap> M12 := MathieuGroup(12);;
gap> Vole.CanonicalImage(M12, s, OnSets);
[ 2, 6, 8, 9, 11, 12 ]
gap> Vole.CanonicalImage(M12, t, OnSets);
[ 2, 6, 8, 9, 11, 12 ]
gap> Vole.CanonicalImage(M12, u, OnSets);
[ 2, 3, 4, 5, 15, 18 ]
```

As s and t have the same canonical image, this means they are in the same orbit of M_{12} , which u is in a different orbit.

How can we find the permutation in M_{12} which maps s to t ? The function `CanonicalImagePerm` gives the permutation which maps something to its canonical image.

Example

```
gap> ps := Vole.CanonicalImagePerm(M12, s, OnSets);
(1,6,9,5,2,12)(3,8,4,11,10,7)
gap> pt := Vole.CanonicalImagePerm(M12, t, OnSets);
```

```

(1,3,4,12,2,6,11,7)(9,10)
gap> OnSets(s, ps); # Gives the canonical image from earlier
[ 2, 6, 8, 9, 11, 12 ]
gap> OnSets(s, ps*(pt^-1)); # Gives t
[ 2, 4, 6, 8, 10, 12 ]
gap> ps*(pt^-1) in M12;
true

```

As we can see here, it is easy to get the canonical image from the result of `CanonicalImagePerm`, but the reverse is not true!

We can find the canonical image of many different types of objects, for example:

```

Example
gap> Vole.CanonicalImage(M12, [[1,2,3,4],[4,5,6,7]], OnSetsSets);
[ [ 2, 3, 7, 12 ], [ 3, 6, 10, 11 ] ]
gap> Vole.CanonicalImage(M12, [[1,2,3,4],[4,5,6,7]], OnSetsTuples);
[ [ 2, 8, 3, 6 ], [ 6, 4, 10, 5 ] ]
gap> Vole.CanonicalImage(M12, DigraphCycle(12), OnDigraphs);
<immutable digraph with 12 vertices, 12 edges>

```

Note that while we could use a tool like Nauty, from the GRAPE or Digraphs packages, to calculate the canonical image of our graph, that doesn't support adding searching within a named group.

The `CanonicalImage` and `CanonicalImagePerm` functions work on a wide range of GAP objects and actions -- if there an object or action you find is not implemented, please report it at [The vole issue tracker](#) and we will investigate if it is possible to add.

3.2 The full Vole interface

We will begin by working through an example of the intersection of a normaliser and a stabiliser, and then look at how this problem can be more efficiently and directly solved in Vole.

Suppose that we wish to compute the intersection of the stabiliser in M_{12} of the set $S := \{1, 2, 4, 5\}$ with the normaliser of H in G , where H and G are defined by the generating sets given below:

```

Example
gap> M12 := MathieuGroup(12);;
gap> S := [ 1, 2, 4, 5 ];;
gap> G := Group([(1,8,7,2,3,10,9,4)(5,6,11,12), (3,11)(4,12,6,8,10)]);;
gap> H := Group([(1,2,3,4,9,10,5,6,11,8)(7,12), (1,5,9)(6,12)(8,10)]);;

```

Thus we wish to compute $\text{Stab}_{M_{12}}(\{1, 2, 3, 5\}) \cap N_G(H)$.

For future reference, we note that the result is a dihedral group with eight elements.

```

Example
gap> answer := Group([(1,2)(3,10)(4,5)(6,7)(8,9)(11,12),
> (2,4)(3,7)(8,12)(9,11)]);
Group([ (1,2)(3,10)(4,5)(6,7)(8,9)(11,12), (2,4)(3,7)(8,12)(9,11) ])
gap> StructureDescription(answer);
"D8"

```

3.2.1 The general GAP approach

With the GAP library, there are several ways to compute this. For example, we can stick as closely to the statement as possible, and first compute the stabiliser and the normaliser separately, and then intersect them:

Example

```
gap> Intersection(Stabiliser(M12, S, OnSets),
>               Normaliser(G, H))
> = answer;
true
```

Or we can slightly reformulate the task, and first intersect M_{12} with G , and then compute the stabiliser of the set in that smaller group, and finally compute the normaliser of H in that (even smaller) stabiliser:

Example

```
gap> M12andG := Intersection(M12, G);;
gap> stab := Stabiliser(M12andG, S, OnSets);;
gap> Normaliser(stab, H) = answer;
true
```

Or we could switch things in the previous example and compute the normaliser before computing the stabiliser in that normaliser:

Example

```
gap> M12andG := Intersection(M12, G);;
gap> norm := Normaliser(M12andG, H);;
gap> Stabiliser(norm, S, OnSets) = answer;
true
```

There are several further combinations. Each of the above strategies can be emulated in Vole by prepending “Vole.” to the beginning of each call to `Stabiliser`, `Normaliser`, and `Intersection`, as described in Chapter 4. However, this is not the recommended approach, as we will see below.

Example

```
gap> stab := Vole.Stabiliser(M12, S, OnSets);;
gap> norm := Vole.Normaliser(G, H);;
gap> answer = Vole.Intersection(stab, norm);
true
```

We quickly realise that solving our problem with the GAP library interface really requires three separate steps to be undertaken in sequence. This raises some potential disadvantages.

- Although each of the above strategies will give the correct answer, it is not obvious which approach gives the best performance: should we compute the normaliser in the stabiliser, or vice versa, or should we do something else? It is not necessarily easy to answer this.
- Breaking the problem up into three separate steps requires three instances of a backtrack search, where each instance is unaware of the ones to come. This is not ideal for a number of reasons.
 - Firstly, backtrack search can be expensive, and so we should aim to minimise the number of times that it is required.
 - Secondly, a search tends to be quicker when there are more ‘restrictions’ on the search space. Therefore, it is typically better to perform one search with many restrictions rather than performing several searches that each have few restrictions.

3.2.2 The general Vole approach

The ‘Vole’ way of solving this problem is to do it in one step with the function `VoleFind.Group` (5.2.2).

For most users, the easiest way to solve the problem with `VoleFind.Group` (5.2.2) is to find a collection of *constraints* that together specify the problem. Constraints are discussed in Chapter 7 of the `BacktrackKit` manual.

Specifically, we are looking for all permutations that are contained in M_{12} , that stabilise the set $\{1, 2, 3, 5\}$, that are also contained in G , and that normalise H . Therefore, we can solve the problem with the following constraints:

Example

```
gap> VoleFind.Group(Constraint.InGroup(M12),
>                  Constraint.Stabilize(S, OnSets),
>                  Constraint.InGroup(G),
>                  Constraint.Normalize(H))
> = answer;
true
```

Vole performs only one search to solve the whole problem.

In order to solve the problem as specified, Vole chooses appropriate refiners for the given collection of constraints. Refiners are documented in Chapter 7. The more confident user may wish to directly specify one or more refiners instead of, and/or in addition to, some of the constraints.

For example, a user may know (or just hope) that the group H is well-suited to the technique that the refiner `GB_Con.NormaliserSimple` from the `GraphBacktracking` package uses to refine for the normaliser. This refiner may be included instead of, or as well as, the constraint `Constraint.Normalize(H)`, since the refiner implies that constraint, but it is perfectly acceptable (and sometimes a good idea) to use multiple refiners for the same constraint.

Example

```
gap> VoleFind.Group(Constraint.InGroup(M12),
>                  Constraint.Stabilize(S, OnSets),
>                  Constraint.InGroup(G),
>                  GB_Con.NormaliserSimple(H))
> = answer;
true
gap> VoleFind.Group(Constraint.InGroup(M12),
>                  Constraint.Stabilize(S, OnSets),
>                  Constraint.InGroup(G),
>                  Constraint.Normalize(H),
>                  GB_Con.NormaliserSimple(H))
> = answer;
true
```

Chapter 4

Emulating traditional interfaces with Vole

4.1 The concept

The functionality of **Vole** overlaps with that of **GAP** and its other packages, such as **images** and **Digraphs**.

Vole has its own native interface, which is described in Chapter 5, and which offers highly configurable access to the underlying graph backtracking algorithm. This is the recommended interface to **Vole** for most users.

However, **Vole** also provides wrappers around its native interface that allow **Vole** to emulate some existing interfaces. The purpose of these wrappers is to lower the ‘barrier to entry’ of **Vole**, so that users who are already familiar with the existing **GAP**/package functions can quickly get started with **Vole**.

Where we identify that **GAP**, or a package, provides a function whose result could reasonably be computed with **Vole**, we provide a wrapper function whose interface closely matches that of the original function, but which uses **Vole** to perform the computation. WE DO NOT CLAIM THAT THE **Vole** WRAPPER FUNCTIONS ARE NECESARILY FASTER THAN THE ORIGINAL FUNCTIONS; see Section 4.1.1.

For example, the **GAP** function **Normaliser** (**Reference:** **Normalizer**) can be used to compute normalisers of permutation groups. Since **Vole** can also be used for such computations, we provide the corresponding function **Vole.Normaliser** (4.4.4), which can be used in the same way. Thus **Normaliser**($\langle A \rangle G \langle /A \rangle$, $\langle A \rangle U \langle /A \rangle$) and **Vole.Normaliser**($\langle A \rangle G \langle /A \rangle$, $\langle A \rangle U \langle /A \rangle$) will (barring bugs!) return equal groups for all permutation groups G and U .

All such functions are contained in the **Vole** (4.2.1) record, which is documented in Section 4.2. The functions themselves are individually documented in Sections 4.4–4.6.

4.1.1 A warning

Vole’s emulated interfaces do not necessarily exhibit the full flexibility, power, and speed of **Vole**. This is especially for those users who are interesting in controlling and specifying the refiners that are used in a given search, for whom the native interface (Chapter 5) is necessary.

In addition, please note that the **Vole** wrapper functions (almost) always use the graph backtracking algorithm, and so they may be significantly slower than the corresponding original functions when

those functions can cleverly simplify the problem (and perhaps even choose a completely different algorithm), based on the properties of the groups and permutations that are involved.

See Section 1.4 for further comments about the relative performance of `Vole` in comparison to other tools.

4.2 The `Vole` record

4.2.1 `Vole`

▷ `Vole`

(global variable)

`Vole` is a record that contains all of the `Vole` wrapper functions that are provided to emulate aspects of `GAP`, and its packages `images` and `Digraphs`. The components of this record are functions that are named to coincide with the corresponding `GAP/images/Digraphs` functions.

For example, `Vole.Normaliser` (4.4.4) emulates `Normaliser` (**Reference: Normalizer**).

Example

```
gap> LoadPackage("vole", false);;
gap> Set(RecNames(Vole));
[ "AutomorphismGroup", "CanonicalDigraph", "CanonicalImage",
  "CanonicalImagePerm", "CanonicalPerm", "Centraliser", "Centralizer",
  "DigraphCanonicalLabelling", "Intersection", "IsConjugate",
  "IsIsomorphicDigraph", "IsomorphismDigraphs", "Normaliser", "Normalizer",
  "RepresentativeAction", "Stabiliser", "Stabilizer", "TwoClosure" ]
```

4.3 The group actions built into `Vole`

The supported combinations of objects and actions are the same for the functions in this chapter that require one or two objects, and a corresponding action. This information is shown in the following table and applies, in particular, to:

- `Vole.Stabiliser` (4.4.2)
- `Vole.RepresentativeAction` (4.4.3)
- `Vole.CanonicalPerm` (4.5.1)
- `Vole.CanonicalImage` (4.5.2)

PERMITTED ACTION	CORRESPONDING OBJECT
<code>OnPoints</code> (Reference: OnPoints) [default]	A positive integer, permutation
<code>OnTuples</code> (Reference: OnTuples)	A list of positive integers
<code>OnSets</code> (Reference: OnSets)	A set of positive integers
<code>OnSetsSets</code> (Reference: OnSetsSets)	A set of sets of positive integers
<code>OnSetsTuples</code> (Reference: OnSetsTuples)	A set of lists of positive integers
<code>OnTuplesSets</code> (Reference: OnTuplesSets)	A list of sets of positive integers
<code>OnTuplesTuples</code> (Reference: OnTuplesTuples)	A list of lists of positive integers
<code>OnDigraphs</code> (Digraphs: OnDigraphs for a digraph and a perm)	A digraph object, or a list of digraphs
<code>OnTuplesDigraphs</code> (Digraphs: OnTuplesDigraphs for a list of digraphs and a perm)	A list of digraphs/adjacency matrices

4.4 Vole functions emulating built-in GAP functions

The following table gives a summary of the Vole wrapper functions and the corresponding GAP functions.

Vole function	Built-in GAP function
Vole.Intersection (4.4.1)	Intersection (Reference: Intersection)
Vole.Stabiliser (4.4.2)	Stabiliser (Reference: Stabilizer)
Vole.RepresentativeAction (4.4.3)	RepresentativeAction (Reference: RepresentativeAction)
Vole.Normaliser (4.4.4)	Normaliser (Reference: Normalizer)
Vole.Centraliser (4.4.5)	Centraliser (Reference: centraliser)
Vole.IsConjugate (4.4.6)	IsConjugate (Reference: IsConjugate)
Vole.TwoClosure (4.4.7)	TwoClosure (Reference: TwoClosure)

4.4.1 Intersection

▷ Vole.Intersection(*U1*, *U2*, ..., *Uk*) (function)

Returns: A perm group, a right coset, or an empty list

Vole.Intersection emulates the built-in GAP operation Intersection (**Reference: Intersection**), and returns the intersection of the group and/or right coset arguments.

If all of the arguments are groups, then Vole.Intersection again returns a group. Otherwise, if the result is nonempty, then Vole.Intersection returns a GAP right coset object. Note that this non-group case differs from GAP's Intersection (**Reference: Intersection**), which always returns a list.

Note that Vole performs the whole intersection in one search, rather than iteratively intersecting the arguments.

There are many reasons why GAP may be faster than Vole for any particular problem; see Section 1.4 for some discussion about this. It may be possible to obtain better performance from Vole by specifying custom refiners with the native interface, see Chapter 5 and Chapter 7.

Example

```
gap> A6 := AlternatingGroup(6);;
gap> D12 := DihedralGroup(IsPermGroup, 12);;
gap> Vole.Intersection(A6, D12);
Group([ (2,6)(3,5), (1,3,5)(2,4,6) ])
gap> Vole.Intersection(A6 * (1,2), D12 * (3,4));
RightCoset(Group([ (2,6)(3,5), (1,3)(4,6) ]), (1,5,4,2,6,3))
gap> Vole.Intersection(A6 * (1,2), D12 * (3,4), PSL(2,5));
[ ]
```

4.4.2 Stabiliser

▷ Vole.Stabiliser(*G*, *object*[, *action*]) (function)

▷ Vole.Stabilizer(*G*, *object*[, *action*]) (function)

Returns: An permutation group

Vole.Stabiliser emulates the built-in GAP operation Stabiliser (**Reference: Stabilizer**).

The *stabiliser* of an *object* in a group *G* under some group *action* is the subgroup of *G* of those elements that fix *object* under *action*, i.e. all permutations *g* in *G* such that $\langle A \rangle \text{action} \langle /A \rangle (\langle A \rangle \text{object} \langle /A \rangle, g) = \langle A \rangle \text{object} \langle /A \rangle$.

The permitted combinations of objects and actions are given in the table below.

The default *action*, in the case that the argument is not given, is `OnPoints` (**Reference: OnPoints**). This is the name in GAP of the action given by the $\sim$ operator, i.e. it corresponds to $\langle A \rangle \text{object} \langle /A \rangle^g$, where g in G . See $\sim$ (**Reference: $\sim$**).

There are many reasons why GAP may be faster than Vole for any particular problem; see Section 1.4 for some discussion about this. It may be possible to obtain better performance from Vole by specifying custom refiners with the native interface, see `VoleFind.Group` (5.2.2) and Chapter 7.

Example

```
gap> Vole.Stabiliser(PGL(2,5), [1,2,3], OnSets);
Group([ (1,3)(5,6), (1,2,3)(4,5,6) ])
gap> D := JohnsonDigraph(4,2);;
gap> G := Stabiliser(PSL(2,5), D, OnDigraphs);;
gap> G = Group([ (1,4,5)(2,6,3), (1,4)(3,6) ]);
true
gap> Elements(G)
> = SortedList(Filtered(PSL(2,5), g -> OnDigraphs(D, g) = D));
true
```

PERMITTED ACTION

`OnPoints` (**Reference: OnPoints**) [default]

`OnTuples` (**Reference: OnTuples**)

`OnSets` (**Reference: OnSets**)

`OnSetsSets` (**Reference: OnSetsSets**)

`OnSetsTuples` (**Reference: OnSetsTuples**)

`OnTuplesSets` (**Reference: OnTuplesSets**)

`OnTuplesTuples` (**Reference: OnTuplesTuples**)

`OnDigraphs` (**Digraphs: OnDigraphs for a digraph and a perm**)

`OnTuplesDigraphs` (**Digraphs: OnTuplesDigraphs for a list of digraphs and a perm**)

CORRESPONDING OBJECT

A positive integer, permutation

A list of positive integers

A set of positive integers

A set of sets of positive integers

A set of lists of positive integers

A list of sets of positive integers

A list of lists of positive integers

A digraph object, or a list of digraphs

A list of digraphs/adjacency matrices

4.4.3 RepresentativeAction

▷ `Vole.RepresentativeAction(G, object1, object2[, action])`

(function)

Returns: A permutation, or fail

`Vole.RepresentativeAction` emulates the built-in GAP function `RepresentativeAction` (**Reference: RepresentativeAction**).

This function returns an element of the permutation group G that maps *object1* to *object2* under the given group *action*, if such an element exists, and it returns `fail` otherwise.

The permitted combinations of objects and actions are given in the table below.

The default *action*, in the case that the argument is not given, is `OnPoints` (**Reference: OnPoints**). This is the name in GAP of the action given by the $\sim$ operator, i.e. it corresponds to $\langle A \rangle \text{object} \langle /A \rangle^g$, where g in G . See $\sim$ (**Reference: $\sim$**).

There are many reasons why GAP may be faster than Vole for any particular problem; see Section 1.4 for some discussion about this. It may be possible to obtain better performance from Vole by specifying custom refiners with the native interface, see `VoleFind.Rep` (5.2.1) and Chapter 7.

Example

```
gap> Vole.RepresentativeAction(SymmetricGroup(4), (1,2,3), (1,2,4));
(1,4,3,2)
```

```

gap> RepresentativeAction(AlternatingGroup(4), (1,2,3), (1,2,4));
fail
gap> D := CycleDigraph(6);;
gap> Vole.RepresentativeAction(PGL(2,5), D, DigraphReverse(D), OnDigraphs);
(1,4)(2,3)(5,6)

```

PERMITTED ACTION	CORRESPONDING OBJECT
OnPoints (Reference: OnPoints) [default]	A positive integer, permutation
OnTuples (Reference: OnTuples)	A list of positive integers
OnSets (Reference: OnSets)	A set of positive integers
OnSetsSets (Reference: OnSetsSets)	A set of sets of positive integers
OnSetsTuples (Reference: OnSetsTuples)	A set of lists of positive integers
OnTuplesSets (Reference: OnTuplesSets)	A list of sets of positive integers
OnTuplesTuples (Reference: OnTuplesTuples)	A list of lists of positive integers
OnDigraphs (Digraphs: OnDigraphs for a digraph and a perm)	A digraph object, or a list of digraphs
OnTuplesDigraphs (Digraphs: OnTuplesDigraphs for a list of digraphs and a perm)	A list of digraphs/adjacency matrices

4.4.4 Normaliser

- ▷ Vole.Normaliser(G , U) (function)
- ▷ Vole.Normalizer(G , U) (function)

Returns: An permutation group

Vole.Normaliser emulates the built-in GAP operation Normaliser (**Reference:** Normalizer).

If G and U are permutation groups, then this function returns the *normaliser* of U in G , $N_G(U)$, which is the stabiliser of U under conjugation by G . If U is instead a permutation, then Vole.Normalizer($\langle A \rangle G \langle A \rangle^{-1}$, $\langle A \rangle U \langle A \rangle^{-1}$) returns $N_G(\langle U \rangle)$.

There are many reasons why GAP may be faster than Vole for any particular problem; see Section 1.4 for some discussion about this. It may be possible to obtain better performance from Vole by specifying custom refiners with the native interface, see VoleFind.Group (5.2.2) and Chapter 7.

Example

```

gap> Vole.Normaliser(SymmetricGroup(6), PSL(2,5)) = PGL(2,5);
true
gap> D12 := DihedralGroup(IsPermGroup, 12);;
gap> Vole.Normaliser(SymmetricGroup(6), (1,2,3,4,5,6)) = D12;
true

```

4.4.5 Centraliser

- ▷ Vole.Centraliser(G , x) (function)
- ▷ Vole.Centralizer(G , x) (function)

Returns: An permutation group

Vole.Centraliser emulates the built-in GAP operation Centraliser (**Reference:** centraliser).

If G is a group and x is a permutation, then this function returns the subgroup of G comprising its elements that commute with x .

If instead x is group, then this function returns the subgroup of G comprising its elements that commute with all elements of x .

Example

```
gap> Vole.Centraliser(MathieuGroup(12), (1,11,9,4,3,2)(5,7,8,6,12,10));
Group([ (1,2,3,4,9,11)(5,10,12,6,8,7), (1,5,3,12,9,8)(2,10,4,6,11,7) ])
gap> Vole.Centraliser(Group((1,2,3,4,5,6)), DihedralGroup(IsPermGroup, 12));
Group([ (1,4)(2,5)(3,6) ])
```

4.4.6 IsConjugate

▷ `Vole.IsConjugate( $G$ ,  $x$ ,  $y$ )` (function)

Returns: true or false

`Vole.IsConjugate` emulates the built-in GAP function `IsConjugate` (**Reference:** `IsConjugate`).

This function returns true if there exists an element of the permutation group G that conjugates x to y , and false otherwise, where x and y are either both permutations, or both permutation groups. Note that x and y are not required to be contained in G .

This function immediately delegates to `Vole.RepresentativeAction` (4.4.3), which finds a representative conjugating element, or proves that none exists.

There are many reasons why GAP may be faster than Vole for any particular problem; see Section 1.4 for some discussion about this. It may be possible to obtain better performance from Vole by specifying custom refiners with the native interface, see `VoleFind.Rep` (5.2.1) and Chapter 7.

Conjugacy of permutations:

Example

```
gap> # Conjugacy of permutations
gap> x := (1,2,3,4,5);; y := (1,2,3,4,6);;
gap> Vole.IsConjugate(SymmetricGroup(6), x, y);
true
gap> Vole.IsConjugate(AlternatingGroup(6), x, y);
false
gap> Vole.IsConjugate(Group([ (5,6) ]), x, y);
true
```

Conjugacy of groups:

Example

```
gap> x := Group([ (1,2,3,4,5) ]);;
gap> y := Group([ (1,2,3,4,6) ]);;
gap> Vole.IsConjugate(SymmetricGroup(6), x, y);
true
gap> Vole.IsConjugate(Group([ (1,2)(3,4) ]), x, y);
false
gap> Vole.IsConjugate(Group([ (5,6) ]), x, y);
true
```

4.4.7 TwoClosure

▷ `Vole.TwoClosure( $G$ )` (function)

Returns: A permutation group

`Vole.TwoClosure` emulates the built-in GAP function `TwoClosure` (**Reference:** `TwoClosure`) for a permutation group.

The *2-closure* of a permutation group G is the largest group whose orbitals (orbits on pairs of positive integers) coincide with those of G ; equivalently, it is the intersection of the automorphism groups of the orbital graphs of G .

WARNING: this function currently requires the `OrbitalGraphs` package, and it will give an error if `OrbitalGraphs` is not yet loaded.

There are many reasons why `GAP` may be faster than `Vole` for any particular problem; see Section 1.4 for some discussion about this. It may be possible to obtain better performance from `Vole` by specifying custom refiners with the native interface, see `VoleFind.Group` (5.2.2) and Chapter 7.

Example

```
gap> LoadPackage("orbitalgraphs", false);;
gap> G := Group([ (1,4)(2,5), (1,3,5)(2,4,6) ]);; # A4 on six points
gap> (3,6) in G;
false
gap> Vole.TwoClosure(G) = ClosureGroup(G, (3,6));
true
```

4.5 Vole functions emulating the images package

The following table gives a summary of the correspondence between `Vole` and the `images` package.

Vole function	images package function
<code>Vole.CanonicalPerm</code> (4.5.1)	<code>CanonicalImagePerm</code> (images: CanonicalPerm)
<code>Vole.CanonicalImage</code> (4.5.2)	<code>CanonicalImage</code> (images: CanonicalImage)

4.5.1 CanonicalPerm

- ▷ `Vole.CanonicalPerm( $G$ ,  $object$  [,  $action$ ])` (function)
- ▷ `Vole.CanonicalImagePerm( $G$ ,  $object$  [,  $action$ ])` (function)

Returns: A permutation

These functions, which are synonyms of each other, emulate `CanonicalImagePerm` (**images: CanonicalImagePerm**) from the `images` package, although they supports a wider range of objects and actions.

Suppose that G is a group, and that $object$ and $object2$ belong to a set on which G has a group $action$. Then `Vole.CanonicalPerm( $\langle A \rangle G \langle /A \rangle$ ,  $\langle A \rangle object \langle /A \rangle$ ,  $\langle A \rangle action \langle /A \rangle$ )` returns an element g of G , and `Vole.CanonicalPerm( $\langle A \rangle G \langle /A \rangle$ ,  $object2$ ,  $\langle A \rangle action \langle /A \rangle$ )` returns an element h of G , such that $object$ and $object2$ are in the same orbit of G under $action$ if and only if $\langle A \rangle object \langle /A \rangle^g = object2^h$.

The permitted combinations of objects and actions are given in the table below.

The default $action$, in the case that the argument is not given, is `OnPoints` (**Reference: OnPoints**). This is the name in `GAP` of the action given by the $\wedge$ operator, i.e. it corresponds to $\langle A \rangle object \langle /A \rangle^g$, where g in G . See $\wedge$ (**Reference: $\wedge$**).

The native `Vole` interface for this kind of computation is provided by `VoleFind.CanonicalPerm` (5.3.2).

WARNING: The permutation given by a canonical search and the canonical image that it defines are NOT GUARANTEED TO BE THE SAME ACROSS DIFFERENT SESSIONS. In particular, canonical permutations/labellings and images may differ in different versions of `Vole`, in different versions of `GAP`, and on different hardware.

Example

```
gap> Vole.CanonicalPerm(PSL(2,5), JohnsonDigraph(4,2), OnDigraphs);
(1,2,6)(3,4,5)
```

PERMITTED ACTION	CORRESPONDING OBJECT
OnPoints (Reference: OnPoints) [default]	A positive integer, permutation
OnTuples (Reference: OnTuples)	A list of positive integers
OnSets (Reference: OnSets)	A set of positive integers
OnSetsSets (Reference: OnSetsSets)	A set of sets of positive integers
OnSetsTuples (Reference: OnSetsTuples)	A set of lists of positive integers
OnTuplesSets (Reference: OnTuplesSets)	A list of sets of positive integers
OnTuplesTuples (Reference: OnTuplesTuples)	A list of lists of positive integers
OnDigraphs (Digraphs: OnDigraphs for a digraph and a perm)	A digraph object, or a list of digraphs
OnTuplesDigraphs (Digraphs: OnTuplesDigraphs for a list of digraphs and a perm)	A list of digraphs/adjacency matrices

4.5.2 CanonicalImage

▷ `Vole.CanonicalImage(G, object [, action])` (function)

Returns: An image of *object*

This function emulates `CanonicalImage` (**images:** `CanonicalImage`) from the `images` package, although it supports a wider range of objects and actions.

`Vole.CanonicalImage` returns `<A>object</A>^g`, where *g* is the permutation returned by `Vole.CanonicalPerm(G, object, action)`. See `Vole.CanonicalPerm` (4.5.1) for more information.

The permitted combinations of objects and actions are given in the table below.

The default *action*, in the case that the argument is not given, is `OnPoints` (**Reference:** `OnPoints`). This is the name in GAP of the action given by the `^` operator, i.e. it corresponds to `<A>object</A>^g`, where *g* in *G*. See `^` (**Reference:** `^`).

The native Vole interface for this kind of computation is provided by `VoleFind.CanonicalPerm` (5.3.2).

WARNING: The permutation given by a canonical search and the canonical image that it defines are NOT GUARANTEED TO BE THE SAME ACROSS DIFFERENT SESSIONS. In particular, canonical permutations/labellings and images may differ in different versions of Vole, in different versions of GAP, and on different hardware.

In the following example, we find that three different tuples of points have two different canonical images in A_5 :

Example

```
gap> tuple1 := [1,2,3,4];; tuple2 := [1,2,3,5];; tuple3 := [1,5,2,3];;
gap> A5 := AlternatingGroup(5);;
gap> Vole.CanonicalImage(A5, tuple1, OnTuples);
[ 5, 4, 3, 2 ]
gap> Vole.CanonicalImage(A5, tuple2, OnTuples);
[ 4, 5, 3, 2 ]
gap> Vole.CanonicalImage(A5, tuple3, OnTuples);
[ 4, 5, 3, 2 ]
```

Therefore, the tuples with the same canonical image are in the same orbit of A_5 , and those with different canonical images are in different orbits:

Example

```
gap> Vole.RepresentativeAction(A5, tuple1, tuple2, OnTuples);
fail
gap> Vole.RepresentativeAction(A5, tuple1, tuple3, OnTuples);
fail
gap> Vole.RepresentativeAction(A5, tuple2, tuple3, OnTuples);
(2,5,3)
```

PERMITTED ACTION	CORRESPONDING OBJECT
OnPoints (Reference: OnPoints) [default]	A positive integer, permutation
OnTuples (Reference: OnTuples)	A list of positive integers
OnSets (Reference: OnSets)	A set of positive integers
OnSetsSets (Reference: OnSetsSets)	A set of sets of positive integers
OnSetsTuples (Reference: OnSetsTuples)	A set of lists of positive integers
OnTuplesSets (Reference: OnTuplesSets)	A list of sets of positive integers
OnTuplesTuples (Reference: OnTuplesTuples)	A list of lists of positive integers
OnDigraphs (Digraphs: OnDigraphs for a digraph and a perm)	A digraph object, or a list of digraphs
OnTuplesDigraphs (Digraphs: OnTuplesDigraphs for a list of digraphs and a perm)	A list of digraphs/adjacency lists

4.6 Vole functions emulating the Digraphs package

The following table gives a summary of the correspondence between Vole and the Digraphs package.

Vole function	Digraphs package function
Vole.AutomorphismGroup (4.6.1)	AutomorphismGroup (Digraphs: AutomorphismGroup for a digraph)
Vole.CanonicalDigraph (4.6.3)	NautyCanonicalDigraph (Digraphs: NautyCanonicalDigraph)
Vole.DigraphCanonicalLabelling (4.6.2)	NautyCanonicalLabelling (Digraphs: NautyCanonicalLabelling)
Vole.IsomorphismDigraphs (4.6.5)	IsomorphismDigraphs (Digraphs: IsomorphismDigraphs for digraphs)
Vole.IsIsomorphicDigraph (4.6.4)	IsIsomorphicDigraph (Digraphs: IsIsomorphicDigraph for digraphs)

4.6.1 AutomorphismGroup

▷ Vole.AutomorphismGroup(*D*) (function)

Returns: A permutation group

Vole.AutomorphismGroup emulates the Digraphs package functions AutomorphismGroup (**Digraphs: AutomorphismGroup for a digraph**), NautyAutomorphismGroup (**Digraphs: NautyAutomorphismGroup**), and BlissAutomorphismGroup (**Digraphs: BlissAutomorphismGroup for a digraph**).

This function returns the automorphism group of the digraph *D*. The *automorphism group* of *D* is the stabiliser of *D* in SymmetricGroup(DigraphVertices(*D*)) under its natural action on digraphs, namely OnDigraphs (**Digraphs: OnDigraphs for a digraph and a perm**). The Vole function Vole.Stabiliser (4.4.2) can be used to compute the stabiliser of *D* in an arbitrary permutation group.

Support for specifying vertex and/or edge colours has not yet been implemented, sorry. We are working on it!

Example

```
gap> Vole.AutomorphismGroup(JohnsonDigraph(4,2));
Group([ (3,4), (2,3,5,4), (1,2,6,5)(3,4) ])
```

4.6.2 DigraphCanonicalLabelling

▷ Vole.DigraphCanonicalLabelling(*D*) (function)

Returns: A permutation

Vole.DigraphCanonicalLabelling emulates the Digraphs package functions BlissCanonicalLabelling (**Digraphs: BlissCanonicalLabelling for a digraph**) and NautyCanonicalLabelling (**Digraphs: NautyCanonicalLabelling for a digraph**).

If *D* is a digraph, and $G = \text{SymmetricGroup}(\text{DigraphVertices}(\langle A \rangle D \langle A \rangle))$, then this function is a shortcut to $\text{Vole.CanonicalPerm}(G, \langle A \rangle D \langle A \rangle, \text{OnDigraphs})$. It is also possible to use $\text{Vole.CanonicalPerm}$ (4.5.1) to canonise *D* in an arbitrary permutation group.

Support for specifying vertex and/or edge colours has not yet been implemented, sorry. We are working on it!

WARNING: The permutation given by a canonical search and the canonical image that it defines are NOT GUARANTEED TO BE THE SAME ACROSS DIFFERENT SESSIONS. In particular, canonical permutations/labellings and images may differ in different versions of Vole, in different versions of GAP, and on different hardware.

Example

```
gap> Vole.DigraphCanonicalLabelling(JohnsonDigraph(4,2));
(1,2,4,5,3,6)
```

4.6.3 CanonicalDigraph

▷ Vole.CanonicalDigraph(*D*) (function)

Returns: A digraph

Vole.CanonicalDigraph emulates the Digraphs package functions BlissCanonicalDigraph (**Digraphs: BlissCanonicalDigraph**) and NautyCanonicalDigraph (**Digraphs: NautyCanonicalDigraph**).

If *D* is a digraph, and $G = \text{SymmetricGroup}(\text{DigraphVertices}(\langle A \rangle D \langle A \rangle))$, then this function is a shortcut to $\text{Vole.CanonicalImage}(G, \langle A \rangle D \langle A \rangle, \text{OnDigraphs})$. It is also possible to use $\text{Vole.CanonicalImage}$ (4.5.2) to canonise *D* in an arbitrary permutation group.

WARNING: The permutation given by a canonical search and the canonical image that it defines are NOT GUARANTEED TO BE THE SAME ACROSS DIFFERENT SESSIONS. In particular, canonical permutations/labellings and images may differ in different versions of Vole, in different versions of GAP, and on different hardware.

Example

```
gap> Vole.CanonicalDigraph(JohnsonDigraph(4,2));
<immutable digraph with 6 vertices, 24 edges>
```

4.6.4 IsIsomorphicDigraph

▷ Vole.IsIsomorphicDigraph(*D1*, *D2*) (function)

Returns: true or false

`Vole.IsIsomorphicDigraph` emulates the `Digraphs` package function `IsIsomorphicDigraph` (**Digraphs: IsIsomorphicDigraph for digraphs**).

If $D1$ and $D2$ are digraphs, then this function returns `true` if there exists any permutation that maps $D1$ to $D2$ under the `OnDigraphs` (**Digraphs: OnDigraphs for a digraph and a perm**) action, and it returns `false` otherwise. See also `IsomorphismDigraphs` (4.6.5), which this function calls.

Example

```
gap> D := CycleDigraph(6);;
gap> Vole.IsIsomorphicDigraph(D, DigraphReverse(D));
true
gap> Vole.IsIsomorphicDigraph(D, DigraphDual(D));
false
```

4.6.5 IsomorphismDigraphs

▷ `Vole.IsomorphismDigraphs( $D1$ ,  $D2$ )`

(function)

Returns: A permutation, or fail

`Vole.IsomorphismDigraphs` emulates the `Digraphs` package function `IsomorphismDigraphs` (**Digraphs: IsomorphismDigraphs for digraphs**).

If $D1$ and $D2$ are digraphs, then this function returns some permutation that maps $D1$ to $D2$ under the `OnDigraphs` (**Digraphs: OnDigraphs for a digraph and a perm**) action, if one exists, and it returns `fail` otherwise. In other words, this function call is equivalent to `VoleFind.Rep(Constraint.Transport(<A>D1</A>, <A>D2</A>, OnDigraphs))`.

`Vole` can also find such a permutation subject to additional properties, such as belonging to particular groups or cosets, or stabilising certain objects. See `VoleFind.Rep` (5.2.1) for more information.

Example

```
gap> D := CycleDigraph(6);;
gap> Vole.IsomorphismDigraphs(D, DigraphReverse(D));
(1,4)(2,3)(5,6)
gap> Vole.IsomorphismDigraphs(D, DigraphDual(D));
fail
```

Chapter 5

The native Vole interface

The native interface to **Vole** is similar to that provided by **ferret**, **BacktrackKit**, and **GraphBacktracking**.

At a basic level, a search can be executed by choosing a suitable list of constraints (and/or refiners, for more expert users) to define the problem to be solved, and then calling the appropriate function on these constraints.

- The name of the function determines the *kind* of search to be executed (whether for a single permutation, or for a group, or for a canonical image, etc). These functions are described in the rest of this chapter.
- Broadly speaking, constraints and/or refiners define properties that together specify the permutations that are valid solutions to the search problem. Constraints and refiners are described in Chapters 6 and 7, respectively.

5.1 The VoleFind record

5.1.1 VoleFind

▷ **VoleFind** (global variable)

VoleFind is a record that contains the functions providing the native interface to **Vole**.

Example

```
gap> LoadPackage("vole", false);;
gap> Set(RecNames(VoleFind));
[ "Canonical", "CanonicalPerm", "Coset", "Group", "Rep", "Representative" ]
```

5.2 Searching for groups, cosets, and representatives with the native interface

5.2.1 VoleFind.Representative

▷ **VoleFind.Representative**(arguments...) (function)

▷ **VoleFind.Rep**(arguments...) (function)

Returns: A permutation, or fail

`VoleFind.Representative` returns a single permutation that satisfies all of the constraints defined by the *arguments*, if one exists, and it returns `fail` otherwise. `VoleFind.Rep` is a synonym for `VoleFind.Representative`.

The *arguments* may be given separately, or as a single list. Each argument may be a **Vole** constraint (Chapter 6), a refiner (Chapter 7; note that a refiner implies a constraint), or one of the following objects:

- A permutation group G , which is interpreted as an instance of the constraint `Constraint.InGroup` (6.4.1) with argument G .
- A **GAP** right coset object U , which is interpreted as an instance of the constraint `Constraint.InCoset` (6.4.2) with argument U .
- A positive integer k , which is interpreted as an instance of the constraint `Constraint.LargestMovedPoint` (6.4.11) with argument k .
- The value `fail`, which is interpreted as an instance of the constraint `Constraint.None` (6.4.15).

For at least one of the *arguments*, **Vole** must be able to immediately deduce a positive integer k , such that for the corresponding constraint:

- there exists a permutation satisfying the constraint if and only if there exists an element of $\text{Sym}([1..k])$ satisfying the constraint.

Otherwise, an error is given. This guarantees that **Vole** terminates (given sufficient resources). See Section 6.1 for examples and further information.

This function supports various options, which are documented in Section 8.2.

Example

```
gap> tuple_transport := Constraint.Transport([1,2,3], [1,2,4], OnTuples);;
gap> VoleFind.Rep(Constraint.InGroup(SymmetricGroup(4)), tuple_transport);
(3,4)
gap> VoleFind.Rep(AlternatingGroup(4), tuple_transport);
fail
```

5.2.2 VoleFind.Group

▷ `VoleFind.Group(arguments...)`

(function)

Returns: A permutation group

`VoleFind.Group` returns the group of permutations that satisfy the constraints defined by the *arguments*. It is assumed, although not verified, that for each such constraint, the set of permutations satisfying it forms a group: this is the responsibility of the user.

The *arguments* may be given separately, or as a single list. Each argument may be a **Vole** constraint (Chapter 6), a refiner (Chapter 7; note that a refiner implies a constraint), or one of the following objects:

- A permutation group G , which is interpreted as an instance of the constraint `Constraint.InGroup` (6.4.1) with argument G .

- A positive integer k , which is interpreted as an instance of the constraint `Constraint.LargestMovedPoint` (6.4.11) with argument k .

For at least one of the *arguments*, **Vole** must be able to immediately deduce a (finite) largest moved point of all the permutations that satisfy the corresponding constraint. Otherwise, an error is given. This guarantees that **Vole** terminates (given sufficient resources). See Section 6.1 for examples and further information.

This function supports various options, which are documented in Section 8.2.

Example

```
gap> graph_auto := Constraint.Stabilise(JohnsonDigraph(4,2), OnDigraphs);;
gap> set_stab := Constraint.Stabilise([2,4,6], OnSets);;
gap> G := VoleFind.Group(graph_auto, set_stab, 6);;
gap> G = Group([ (2,4)(3,5), (1,3,5)(2,6,4) ]);
true
```

Note that multiple groups-by-generators may be given as constraints:

Example

```
gap> norm_PSL25 := Constraint.Normalise(PSL(2,5));;
gap> in_A6 := Constraint.InGroup(AlternatingGroup(6));;
gap> in_D12 := Constraint.InGroup(DihedralGroup(IsPermGroup, 12));;
gap> G := VoleFind.Group(in_A6, in_D12, norm_PSL25);;
gap> G = Group([ (1,3,5)(2,4,6) ]);
true
```

5.2.3 VoleFind.Coset

▷ `VoleFind.Coset(arguments...)`

(function)

Returns: A GAP right coset of a permutation group, or `fail`

For this function, it is assumed, although not verified, that for each constraint defined by the arguments, the set of permutations satisfying it either is empty, or forms a right coset of some group. It is the responsibility of the user to ensure that this is the case.

Given this, `VoleFind.Coset` returns the set of permutations that satisfy the constraints defined by the arguments, in the case that the set is nonempty, and it returns `fail` otherwise. The set of permutations is returned as a GAP right coset object.

The arguments may be given separately, or as a single list. Each argument may be a **Vole** constraint (Chapter 6), a refiner (Chapter 7; note that a refiner implies a constraint), or one of the following objects:

- A permutation group G , which is interpreted as an instance of the constraint `Constraint.InGroup` (6.4.1) with argument G .
- A GAP right coset object U , which is interpreted as an instance of the constraint `Constraint.InCoset` (6.4.2) with argument U .
- A positive integer k , which is interpreted as an instance of the constraint `Constraint.LargestMovedPoint` (6.4.11) with argument k .
- The value `fail`, which is interpreted as an instance of the constraint `Constraint.None` (6.4.15).

For at least one of the *arguments*, **Vole** must be able to immediately deduce a (finite) largest moved point of all the permutations that satisfy the corresponding constraint. Otherwise, an error is given. This guarantees that **Vole** terminates (given sufficient resources). See Section 6.1 for examples and further information.

This function supports various options, which are documented in Section 8.2.

Example

```
gap> tuple_transport := Constraint.Transport([1,2,3], [1,2,4], OnTuples);
gap> VoleFind.Coset(Constraint.InGroup(SymmetricGroup(6)), tuple_transport);
RightCoset(Group([ (5,6), (4,5,6) ]), (3,4,6))
gap> VoleFind.Coset(AlternatingGroup(4), tuple_transport);
fail
gap> VoleFind.Coset(AlternatingGroup(5), Constraint.Transport(
> CycleDigraph(5), DigraphReverse(CycleDigraph(5)), OnDigraphs));
RightCoset(Group([ (1,2,3,4,5) ]), (1,4)(2,3))
```

5.3 Canonising with the native **Vole** interface

Given a group G , a set X , and an action of G on X , then a *canoniser* for X with respect to the action of G is a function f from X to itself such that:

- for all $x \in X$, x and $f(x)$ are in the same orbit of G on X ; and
- for all $x, y \in X$, $f(x) = f(y)$ if and only if x and y are in the same orbit of G on X .

In other words, a canoniser maps a point $x \in X$ to a canonical element of its orbit under the action of G .

This canonical element is known as the *canonical image* of x with respect to the action of G . It is sometimes called the *canonical form* of X .

In this context, a *canonical permutation* for x is any element of G that maps x to its canonical image under the action of G on x . This is sometimes called a *canonical labelling*. Note that there is not necessarily a unique canonical permutation for each element of X . Indeed, by the orbit-stabiliser theorem, the number of canonical permutations for $x \in X$ is equal to the size of the stabiliser of x in G .

5.3.1 VoleFind.Canonical

▷ **VoleFind.Canonical**(G , *arguments...*) (function)

Returns: A record

VoleFind.Canonical is the main function in **Vole** for canonising various kinds of objects, i.e. finding canonical images and permutations. See the beginning of Section 5.3 for a definition of these terms.

The forthcoming details are crucial for obtaining meaningful results from **VoleFind.Canonical**.

*For users who wish to canonise an object under an action listed in the table in **Constraint.Stabilise** (6.4.6), and who do not wish to specify particular refiners, it may be easier to use the simpler functions **Vole.CanonicalPerm** (4.5.1) and **Vole.CanonicalImage** (4.5.2).*

The first argument to **VoleFind.Canonical** must be the group G in which to canonise. The remaining *arguments* specify the rest of the canonisation problem; in particular, they define the relevant object and action.

The *arguments* that follow G may be given separately, or in a list. Each of the *arguments* must be an instance of a `Constraint.Stabilise` (6.4.6) constraint, either directly, or indirectly as an instance of `Constraint.Normalise` (6.4.7) or `Constraint.Centralise` (6.4.8). NOTE THAT IT IS NOT PERMITTED TO INCLUDE CONSTRAINTS OF THE KIND PRODUCED BY `Constraint.InGroup` (6.4.1).

It is also permitted to include special kinds of refiners as arguments to `VoleFind.Canonical`, although we do not yet document the details of this, since the theory and implementation is still under active development. (Refiners will be documented in Chapter 7).

For the following, we will suppose that *arguments...* is a list of k constraints `Constraint.Stabilise(object_i,action_i)`, for $i=1..k$ in sequence. Then `VoleFind.Canonical` canonises the k -tuple $[object_1, \dots, object_k]$, where the action on the i -th coordinate is *action_i*.

In order to canonise in G another k -tuple of the same kind, such as $[nextobject_1, \dots, nextobject_k]$ with respect to the same action, and in a way that is comparable to the first canonisation, it is necessary to call `VoleFind.Canonical` with the same group G followed by the *arguments...* `Constraint.Stabilise(nextobject_i,action_i)`, IN THE SAME ORDER.

The result of `VoleFind.Canonical` is given as a record, with the following components (see the beginning of Section 5.3 for a definition of some of the following terms):

- **canonical**: a canonical permutation in G for the object, with respect to the action.
- **group**: the stabiliser of the object in G , under the action. This is computed as a by-product of canonisation.

This function supports various options, which are documented in Section 8.2.

WARNING: The permutation given by a canonical search and the canonical image that it defines are NOT GUARANTEED TO BE THE SAME ACROSS DIFFERENT SESSIONS. In particular, canonical permutations/labellings and images may differ in different versions of *Vole*, in different versions of *GAP*, and on different hardware. In addition, please note that the result also depends on order in which the *arguments* are given, and on the specific arguments that are used.

In the following examples, we first show how to canonise two cycle digraphs in the alternating group of degree 6, to find that they are indeed in the same orbit of A_6 under the natural action. We canonise them again in a different group, but this time as *vertex-coloured* digraphs, by canonising them digraph simultaneously with a corresponding colouring of the vertices.

Example

```
gap> cycle := CycleDigraph(6);;
gap> reverse := DigraphReverse(cycle);;
gap> A6 := AlternatingGroup(6);;
gap> canon1 := VoleFind.Canonical(A6, Constraint.Stabilise(cycle, OnDigraphs));
rec( canonical := (1,3,4,6,5), group := Group([ (1,3,5)(2,4,6) ]) )
gap> canon2 := VoleFind.Canonical(A6,
> Constraint.Stabilise(reverse, OnDigraphs));
rec( canonical := (1,4,5), group := Group([ (1,3,5)(2,4,6) ]) )
```

Let us verify that the canonical permutations are indeed in A_6 , and that the group record component is indeed the stabiliser in A_6 :

Example

```
gap> SignPerm(canon1.canonical) = 1 and SignPerm(canon2.canonical) = 1
> and canon1.group = Vole.Stabiliser(A6, cycle, OnDigraphs)
```

```
> and canon2.group = Vole.Stabiliser(A6, reverse, OnDigraphs);
true
```

Next, let's compare the canonical images of the digraphs, to test whether they are in the same orbit of A_6 . et us also verify this result with `Vole.RepresentativeAction` (4.4.3):

Example

```
gap> OnDigraphs(cycle, canon1.canonical)
> = OnDigraphs(reverse, canon2.canonical);
true
gap> Vole.RepresentativeAction(A6, cycle, reverse, OnDigraphs);
(1,5)(2,4)
```

Next, we turn to vertex-coloured digraphs. This time, we will canonise in the 2-transitive group G defined below. We first verify that `cycle` and `reverse` are in the same orbit of G , as digraphs:

Example

```
gap> G := Group([ (1,2,3,4,6), (1,4)(5,6) ]);
gap> Vole.RepresentativeAction(G, cycle, reverse, OnDigraphs) <> fail;
true
```

We will colour the vertices 1, 3, and 5 of `cycle` with colour 1, and the remainder with colour 2; and we will colour the vertices of `reverse` in the opposite way.

Example

```
gap> colours1 := [[1,3,5],[2,4,6]];
gap> colours2 := [[2,4,6],[1,3,5]];
```

We may therefore consider a vertex-coloured digraph as a pair `[digraph, colours]`, with `colours` in the above form, and a permutation group acts on vertex-coloured digraphs by acting via `OnDigraphs` on the first component, and acting via `OnTuplesSets` on the second component.

Example

```
gap> canon1 := VoleFind.Canonical(G,
>                               Constraint.Stabilise(cycle, OnDigraphs),
>                               Constraint.Stabilise(colours1, OnTuplesSets));
rec( canonical := (1,5,2,3,6), group := Group(()) )
gap> canon2 := VoleFind.Canonical(G,
>                               Constraint.Stabilise(reverse, OnDigraphs),
>                               Constraint.Stabilise(colours2, OnTuplesSets));
rec( canonical := (1,6,5,4,3), group := Group(()) )
```

We find that these vertex-coloured digraphs are not in the same orbit of G :

Example

```
gap> OnDigraphs(cycle, canon1.canonical)
> = OnDigraphs(reverse, canon2.canonical);
false
```

However, if we canonise them again in the whole symmetric group of degree 6, we find that they *are* in the same orbit of it as coloured digraphs:

Example

```
gap> canon1 := VoleFind.Canonical(SymmetricGroup(6),
>                               Constraint.Stabilise(cycle, OnDigraphs),
>                               Constraint.Stabilise(colours1, OnTuplesSets));
```

```

rec( canonical := (1,5,2,3,4,6), group := Group([ (1,3,5)(2,4,6) ]) )
gap> canon2 := VoleFind.Canonical(SymmetricGroup(6),
>                               Constraint.Stabilise(reverse, OnDigraphs),
>                               Constraint.Stabilise(colours2, OnTuplesSets));
rec( canonical := (1,3)(2,5,6,4), group := Group([ (1,3,5)(2,4,6) ]) )
gap> OnDigraphs(cycle, canon1.canonical)
> = OnDigraphs(reverse, canon2.canonical);
true

```

5.3.2 VoleFind.CanonicalPerm

▷ `VoleFind.CanonicalPerm( $G$ , arguments...)` (function)

Returns: A permutation

This function returns `VoleFind.Canonical(<A>G</A>, <A>arguments...</A>).canonical`. This is the canonical component of the record returned by `VoleFind.Canonical` (5.3.1), under the same arguments.

Please see the documentation of `VoleFind.Canonical` (5.3.1) for much more information.

WARNING: The permutation given by a canonical search and the canonical image that it defines are NOT GUARANTEED TO BE THE SAME ACROSS DIFFERENT SESSIONS. In particular, canonical permutations/labellings and images may differ in different versions of **Vole**, in different versions of **GAP**, and on different hardware. In addition, please note that the result also depends on order in which the *arguments* are given, and on the specific arguments that are used.

The following example shows how to compute a canonical permutation for the group $\langle(12)\rangle$ under conjugation by A_4 :

Example

```

gap> VoleFind.CanonicalPerm(AlternatingGroup(4),
> Constraint.Normalise(Group([ (1,2) ]))
> );
(1,4)(2,3)

```

Thus the canonical image of $\langle(12)\rangle$ under this action of A_4 is the group $\langle(12)\rangle^{(14)(23)}$, i.e. $\langle(34)\rangle$.

This second example shows how to compute a canonical permutation for the pair $[S, D]$ under the specified componentwise action of S_4 , where S is the set-of-sets $\{\{1,2\}, \{1,4\}, \{2,3\}, \{3,4\}\}$, and D is the cycle digraph on four vertices:

Example

```

gap> VoleFind.CanonicalPerm(SymmetricGroup(4),
> Constraint.Stabilise([ [1,2], [1,4], [2,3], [3,4] ], OnSetsSets),
> Constraint.Stabilise(CycleDigraph(4), OnDigraphs)
> );
(1,2,3)

```

Chapter 6

Constraints

6.1 Bounds associated with a constraint or refiner

In **GAP**, permutations are defined on the set of all positive integers (although each permutation may move only a finite set of points, and there is a system-dependent maximum point that is allowed to be moved).

Vole can only search within a concrete finite symmetric group. Therefore, when giving **Vole** a collection of constraints that define a search problem, the search space must be bounded. More specifically, **Vole** must be easily able to deduce a positive integer k such that the whole search can take place within $\text{Sym}([1..k])$. This guarantees that **Vole** will terminate (given sufficient resources).

To help **Vole** make such a deduction, each constraint and refiner is associated with the following values: a *largest moved point*, and a *largest required point*.

Any call to `VoleFind.Group` (5.2.2) or `VoleFind.Coset` (5.2.3) requires at least one constraint that defines a *finite* largest moved point, and any call to `VoleFind.Representative` (5.2.1) requires at least one constraint that defines a finite largest required point or a finite largest moved point.

LARGEST MOVED POINT

The largest *moved* point of a constraint is either *infinity*, or a positive integer k for which it is known a priori that any permutation satisfying the constraint fixes all points strictly greater than k .

For example, the largest moved point of the constraint `Constraint.InGroup(G)` is `LargestMovedPoint(G)`, see `LargestMovedPoint` (**Reference: LargestMovedPoint for a list or collection of permutations**). On the other hand, any permutation stabilises the empty set, so there is not largest moved point of the constraint `Constraint.Stabilise([], OnSets)`; therefore the value in this case must be *infinity*.

LARGEST REQUIRED POINT

The largest *required* point of a constraint is either *infinity*, or a positive integer k such that there exists a permutation satisfying the constraint if and only if there exists a permutation in $\text{Sym}([1..k])$ satisfying the constraint.

For example, if `set` is a set of positive integers, then the largest required point of the constraint `Constraint.Stabilise(set, OnSets)` is `Maximum(set)`.

The largest moved point of a constraint can serve as an upper bound for the largest required point of a constraint.

Example

```
gap> LoadPackage("vole", false);;
```

6.2 The concept of constraints

Fundamentally, the partition backtrack algorithm (and its generalisations) performs a search for permutations that satisfy a collection of constraints.

A *constraint* is a true/false mathematical property of permutations, such that if the set of permutations satisfying the property is nonempty, then that set must be a (possibly infinite) permutation group, or a coset thereof. For constraints to be useful in practice, it should be ‘easy’ to test whether any given permutation satisfies the property.

For example:

- “is a member of the group $G = \langle X \rangle$ ”,
- “transports the set A to the set B”,
- “commutes with the permutation x ”,
- “conjugates the group $G = \langle X \rangle$ to the group $H = \langle Y \rangle$ ”,
- “is an automorphism of the graph Γ ”, and
- “is even”

are all examples of constraints. On the other hand:

- “is a member of the socle of the group G ”, and
- “is a member of a largest maximal subgroup of the group G ”

do not qualify, unless generating sets for the socle and the largest maximal subgroups of G are *already* known, and there is a unique such maximal subgroup (in which case these properties become instances of the constraint “is a member of the group defined by the generating set...”).

The term ‘constraint’ comes from the computer science field of constraint satisfaction problems, constraint programming, and constraint solvers, with which backtrack search algorithms are very closely linked.

A number of built in constraints, and the functions to create them, are contained in the `Constraint` (6.3.1) record. The members of this record are documented individually in Section 6.4.

To perform a search, it is necessary to (at least implicitly) specify constraints that, in conjunction, define the permutation(s) that you wish to find. A constraint will typically be converted into one or more *refiners* by that the time that a search takes place. Refiners are introduced in Chapter 7, which are the low-level code which implement constraints. We do not explicitly document the conversion of constraints into refiners; the conversion may change in the future.

6.3 The Constraints record

6.3.1 Constraint

▷ `Constraint`

(global variable)

`Constraint` is a record that contains functions for producing all of the constraints provided by default.

The members of `Constraint` are documented individually in Section 6.4.

The members whose names differ only by their “-ise” and “-ize” endings are synonyms, included to accommodate different spellings in English.

Example

```
gap> LoadPackage("BacktrackKit", false);
gap> Set(RecNames(Constraint));
[ "Centralise", "Centralize", "Conjugate", "Everything", "InCoset",
  "InGroup", "InLeftCoset", "InRightCoset", "IsEven", "IsOdd", "IsTrivial",
  "LargestMovedPoint", "MovedPoints", "None", "Normalise", "Normalize",
  "Nothing", "Stabilise", "Stabilize", "Transport" ]
```

6.4 Constraints via the `Constraint` record

In this section, we individually document the functions of the `Constraint` (6.3.1) record, which can be used to create the built-in constraints provided by `BacktrackKit`.

Many of these constraints come in pairs, with a “group” version, and a corresponding “coset” version. These relationships are given in the following table.

Group version	Coset version
<code>Constraint.InGroup</code> (6.4.1)	<code>Constraint.InCoset</code> (6.4.2) <code>Constraint.InRightCoset</code> (6.4.3) <code>Constraint.InLeftCoset</code> (6.4.4)
<code>Constraint.Stabilise</code> (6.4.6)	<code>Constraint.Transport</code> (6.4.5)
<code>Constraint.Normalise</code> (6.4.7)	<code>Constraint.Conjugate</code> (6.4.9)
<code>Constraint.Centralise</code> (6.4.8)	<code>Constraint.Conjugate</code> (6.4.9)
<code>Constraint.MovedPoints</code> (6.4.10)	N/A
<code>Constraint.LargestMovedPoint</code> (6.4.11)	N/A
<code>Constraint.IsEven</code> (6.4.12)	<code>Constraint.IsOdd</code> (6.4.13)
<code>Constraint.IsTrivial</code> (6.4.14)	N/A
N/A	<code>Constraint.None</code> (6.4.15)

6.4.1 `Constraint.InGroup`

▷ `Constraint.InGroup( $G$ )` (function)

Returns: A constraint

This constraint is satisfied by precisely those permutations in the permutation group G .

Example

```
gap> con1 := Constraint.InGroup(DihedralGroup(IsPermGroup, 8));
<constraint: in group: Group( [ (1,2,3,4), (2,4) ] )>
gap> con2 := Constraint.InGroup(AlternatingGroup(4));
<constraint: in group: AlternatingGroup( [ 1 .. 4 ] )>
```

6.4.2 `Constraint.InCoset`

▷ `Constraint.InCoset( $U$ )` (function)

Returns: A constraint

This constraint is satisfied by precisely those permutations in the GAP right coset object U .

See also `Constraint.InLeftCoset` (6.4.4) and `Constraint.InRightCoset` (6.4.3), which allow a coset to be specified by a subgroup and a representative element.

Example

```
gap> U := PSL(2,5) * (3,4,6);
RightCoset(Group([ (3,5)(4,6), (1,2,5)(3,4,6) ]),(3,4,6))
gap> Constraint.InCoset(U);
<constraint: in coset: Group( [ (3,5)(4,6), (1,2,5)(3,4,6) ] ) * (3,4,6)
```

6.4.3 Constraint.InRightCoset

▷ `Constraint.InRightCoset(G, x)` (function)

Returns: A constraint

This constraint is satisfied by precisely those permutations in the right coset of the group G determined by the permutation x .

See also `Constraint.InLeftCoset` (6.4.4) for the left-hand version, and `Constraint.InCoset` (6.4.2) for a GAP right coset object.

Example

```
gap> Constraint.InRightCoset(PSL(2,5), (3,4,6));
<constraint: in coset: Group( [ (3,5)(4,6), (1,2,5)(3,4,6) ] ) * (3,4,6)
```

6.4.4 Constraint.InLeftCoset

▷ `Constraint.InLeftCoset(G, x)` (function)

Returns: A constraint

This constraint is satisfied by precisely those permutations in the left coset of the group G determined by the permutation x .

See also `Constraint.InRightCoset` (6.4.3) for the right-hand version, and `Constraint.InCoset` (6.4.2) for a GAP right coset object.

Example

```
gap> Constraint.InLeftCoset(PSL(2,5), (3,4,6));
<constraint: in coset: Group( [ (3,6)(4,5), (1,2,5)(3,4,6) ] ) * (3,4,6)
```

6.4.5 Constraint.Transport

▷ `Constraint.Transport(object1, object2[, action])` (function)

Returns: A constraint

This constraint is satisfied by precisely those permutations that map *object1* to *object2* under the given group *action*, i.e. all permutations g such that $\langle A \rangle \text{action} \langle A \rangle (\langle A \rangle \text{object1} \langle A \rangle, g) = \langle A \rangle \text{object2} \langle A \rangle$. Note that the set of such permutations may be infinite.

The combinations of objects and actions that are supported by `Constraint.Transport` are given in the table below.

The default *action*, in the case that the argument is not given, is `OnPoints` (**Reference: OnPoints**). This is the name in GAP of the action given by the $\wedge$ operator, i.e. it corresponds to $\langle A \rangle \text{object} \langle A \rangle \wedge g$, where g in G . See $\wedge$ (**Reference: $\wedge$**).

PERMITTED ACTION	CORRESPONDING OBJECT
OnPoints (Reference: OnPoints) [default]	A positive integer, permutation
OnTuples (Reference: OnTuples)	A list of positive integers
OnSets (Reference: OnSets)	A set of positive integers
OnSetsSets (Reference: OnSetsSets)	A set of sets of positive integers
OnSetsTuples (Reference: OnSetsTuples)	A set of lists of positive integers
OnTuplesSets (Reference: OnTuplesSets)	A list of sets of positive integers
OnTuplesTuples (Reference: OnTuplesTuples)	A list of lists of positive integers
OnDigraphs (Digraphs: OnDigraphs for a digraph and a perm)	A digraph object, or a list of digraphs
OnTuplesDigraphs (Digraphs: OnTuplesDigraphs for a list of digraphs and a perm)	A list of digraphs/adjacency lists

Example

```
gap> setofsets1 := [[1, 3, 6], [2, 3, 6]];;
gap> setofsets2 := [[1, 2, 5], [1, 5, 7]];;
gap> con := Constraint.Transport(setofsets1, setofsets2, OnSetsSets);
<constraint: transporter of [ [ 1, 3, 6 ], [ 2, 3, 6 ] ] to [ [ 1, 2, 5 ], [ 1, 5, 7 ] ] under OnSetsSets>
```

6.4.6 Constraint.Stabilise

- ▷ Constraint.Stabilise(*object* [, *action*]) (function)
 ▷ Constraint.Stabilize(*object* [, *action*]) (function)

Returns: A constraint

This constraint is satisfied by precisely those permutations that fix *object* under the given group *action*, i.e. all permutations g such that $\langle A \rangle \text{action} \langle A \rangle (\langle A \rangle \text{object} \langle A \rangle, g) = \langle A \rangle \text{object} \langle A \rangle$. Note that the set of such permutations may be infinite.

The combinations of objects and actions that are supported by Constraint.Stabilise are given in the table below.

The default *action*, in the case that the argument is not given, is OnPoints (**Reference:** OnPoints). This is the name in GAP of the action given by the $\wedge$ operator, i.e. it corresponds to $\langle A \rangle \text{object} \langle A \rangle^\wedge g$, where g in G . See $\wedge$ (**Reference:** $\wedge$).

PERMITTED ACTION	CORRESPONDING OBJECT
OnPoints (Reference: OnPoints) [default]	A positive integer, permutation
OnTuples (Reference: OnTuples)	A list of positive integers
OnSets (Reference: OnSets)	A set of positive integers
OnSetsSets (Reference: OnSetsSets)	A set of sets of positive integers
OnSetsTuples (Reference: OnSetsTuples)	A set of lists of positive integers
OnTuplesSets (Reference: OnTuplesSets)	A list of sets of positive integers
OnTuplesTuples (Reference: OnTuplesTuples)	A list of lists of positive integers
OnDigraphs (Digraphs: OnDigraphs for a digraph and a perm)	A digraph object, or a list of digraphs
OnTuplesDigraphs (Digraphs: OnTuplesDigraphs for a list of digraphs and a perm)	A list of digraphs/adjacency lists

Example

```
gap> con1 := Constraint.Stabilise(CycleDigraph(6), OnDigraphs);
<constraint: stabiliser of CycleDigraph(6) under OnDigraphs>
gap> con2 := Constraint.Stabilise([2,4,6], OnSets);
<constraint: stabiliser of [ 2, 4, 6 ] under OnSets>
```

6.4.7 Constraint.Normalise

- ▷ `Constraint.Normalise( $G$ )` (function)
- ▷ `Constraint.Normalize( $G$ )` (function)

Returns: A constraint

This constraint is satisfied by precisely those permutations that normalise the permutation group G , i.e. that preserve G under conjugation.

Note that the set of such permutations is infinite.

Example

```
gap> Constraint.Normalise(PSL(2,5));
<constraint: normalise Group( [ (3,5)(4,6), (1,2,5)(3,4,6) ] )>
```

6.4.8 Constraint.Centralise

- ▷ `Constraint.Centralise( $G$ )` (function)
- ▷ `Constraint.Centralize( $G$ )` (function)

Returns: A constraint

This constraint is satisfied by precisely those permutations that commute with G , if G is a permutation, or that commute with every element of G , if G is a permutation group.

Note that the set of such permutations is infinite.

Example

```
gap> D12 := DihedralGroup(IsPermGroup, 12);;
gap> Constraint.Centralise(D12);
<constraint: centralise group Group( [ (1,2,3,4,5,6), (2,6)(3,5) ] )>
gap> x := (1,6)(2,5)(3,4);;
gap> Constraint.Centralise(x);
<constraint: centralise perm (1,6)(2,5)(3,4)>
```

6.4.9 Constraint.Conjugate

- ▷ `Constraint.Conjugate( $x$ ,  $y$ )` (function)

Returns: A constraint

This constraint is satisfied by precisely those permutations that conjugate x to y , where x and y are either both permutations, or both permutation groups.

Note that the set of such permutations may be infinite.

This constraint is equivalent to `Constraint.Transport(<A> $x$ </A>, <A> $y$ </A>, OnPoints)`.

Example

```
gap> Constraint.Conjugate((3,4)(2,5,1), (1,2,3)(4,5));
<constraint: conjugate perm (1,2,5)(3,4) to (1,2,3)(4,5)>
```

6.4.10 Constraint.MovedPoints

- ▷ `Constraint.MovedPoints( $pointlist$ )` (function)

Returns: A constraint

This constraint is a shorthand for `Constraint.InGroup(SymmetricGroup(<A> $pointlist$ </A>))`. See `Constraint.InGroup` (6.4.1).

Example

```
gap> con1 := Constraint.MovedPoints([1..5]);
<constraint: moved points: [ 1 .. 5 ]>
gap> con2 := Constraint.MovedPoints([2,6,4,5]);
<constraint: moved points: [ 2, 6, 4, 5 ]>
```

6.4.11 Constraint.LargestMovedPoint

▷ `Constraint.LargestMovedPoint(point)` (function)

Returns: A constraint

This constraint is a shorthand for `Constraint.InGroup(SymmetricGroup(<A>point</A>))`, where *point* is a nonnegative integer. See `Constraint.InGroup` (6.4.1).

Example

```
gap> con := Constraint.LargestMovedPoint(5);
<constraint: largest moved point: 5>
```

6.4.12 Constraint.IsEven

▷ `Constraint.IsEven` (global variable)

This constraint is satisfied by the even permutations, i.e. those permutations with sign 1. In other words, this constraint restricts a search to some alternating group.

Note that the set of such permutations is infinite.

Example

```
gap> Constraint.IsEven;
<constraint: is even permutation>
gap> Representative(Constraint.IsEven);
()
```

6.4.13 Constraint.IsOdd

▷ `Constraint.IsOdd` (global variable)

This constraint is satisfied by the odd permutations, i.e. those permutations with sign -1. In other words, this constraint restricts a search to the unique coset of some alternating group.

Note that the set of such permutations is infinite.

Example

```
gap> Constraint.IsOdd;
<constraint: is odd permutation>
gap> Representative(Constraint.IsOdd);
(1,2)
```

6.4.14 Constraint.IsTrivial

▷ `Constraint.IsTrivial` (global variable)

This constraint is satisfied by the identity permutation and no others.

This constraint will typically not be required by the user.

Example

```
gap> Constraint.IsTrivial;  
<trivial constraint: is identity permutation>  
gap> Representative(Constraint.IsTrivial);  
( )
```

6.4.15 Constraint.None

▷ Constraint.None

(global variable)

This constraint is satisfied by no permutations.
This constraint will typically not be required by the user.

Example

```
gap> Constraint.None;  
<empty constraint: satisfied by no permutations>  
gap> Representative(Constraint.None);  
fail
```

6.4.16 Constraint.Everything

▷ Constraint.Everything

(global variable)

This constraint is satisfied by all permutations.
This constraint will typically not be required by the user.

Example

```
gap> Constraint.Everything;  
<constraint: satisfied by all permutations>  
gap> Representative(Constraint.Everything);  
( )
```

Chapter 7

Refiners in Vole

7.1 Refiners

Refiners in Vole are still a work in progress, and are not yet properly documented. Please check back in the next version. appropriate refiners for the given constraints.

7.2 The VoleRefiner record

7.2.1 VoleRefiner

▷ VoleRefiner (global variable)

VoleRefiner is a record that contains all of the refiners that are included in Vole.
GraphBacktracking and BacktrackKit refiners are also compatible with Vole.

Example

```
gap> LoadPackage("vole", false);
gap> Set(RecNames(VoleRefiner));
[ "DigraphStab", "DigraphTransporter", "FromConstraint", "InSymmetricGroup",
  "SetSetStab", "SetSetTransporter", "SetStab", "SetTransporter",
  "SetTupleStab", "SetTupleTransporter", "TupleStab", "TupleTransporter" ]
```

7.3 Vole refiners via the VoleRefiner record

7.3.1 VoleRefiner.InSymmetricGroup

▷ VoleRefiner.InSymmetricGroup(x) (function)

Returns: A Vole refiner
Something

Example

```
gap> true;
true
```

7.3.2 VoleRefiner.SetStab

- ▷ `VoleRefiner.SetStab(s)` (function)
 - ▷ `VoleRefiner.SetTransporter(s, t)` (function)
- Returns:** A Vole refiner
Something

Example

```
gap> true;
true
```

7.3.3 VoleRefiner.TupleStab

- ▷ `VoleRefiner.TupleStab(s)` (function)
 - ▷ `VoleRefiner.TupleTransporter(s, t)` (function)
- Returns:** A Vole refiner
Something

Example

```
gap> true;
true
```

7.3.4 VoleRefiner.SetSetStab

- ▷ `VoleRefiner.SetSetStab(s)` (function)
 - ▷ `VoleRefiner.SetSetTransporter(s, t)` (function)
- Returns:** A Vole refiner
Something

Example

```
gap> true;
true
```

7.3.5 VoleRefiner.SetTupleStab

- ▷ `VoleRefiner.SetTupleStab(s)` (function)
 - ▷ `VoleRefiner.SetTupleTransporter(s, t)` (function)
- Returns:** A Vole refiner
Something

Example

```
gap> true;
true
```

7.3.6 VoleRefiner.DigraphStab

- ▷ `VoleRefiner.DigraphStab(s)` (function)
 - ▷ `VoleRefiner.DigraphTransporter(s, t)` (function)
- Returns:** A Vole refiner
Something

Example

```
gap> true;
true
```

7.4 Choosing a refiner for a given constraint

7.4.1 VoleRefiner.FromConstraint

- ▷ `VoleRefiner.FromConstraint(constraint)` (function)
- Returns:** A Vole, GraphBacktracking, or BacktrackKit refiner
Something.

Example

```
gap> true;  
true
```

Chapter 8

Expert use of Vole

8.1 InfoVole – getting extra information about a Vole search

8.1.1 InfoVole

▷ InfoVole

(info class)

InfoVole is the primary info class for Vole. See **(Reference: Info Functions)** for a description of info classes in GAP.

The default info level is 0. Most info messages are given at level 2, but some messages are given at other levels, up to level 4.

8.2 Options with the native Vole interface

Most Vole functions support several options, which are implemented with the GAP options system, which is described in the GAP reference manual; see **(Reference: Options Stack)**.

In brief, options can be set for a function call by first giving the arguments, then writing a colon, and then assigning values to some or all of the permitted option names. Writing an option name without an explicitly assigned value is a shorthand for assigning the value `true` to that option. Note that *options are passed down to subsequent function calls* that are initiated by the original call.

Here is an example of a Vole function call with the option `raw` set to `true`:

Example

```
gap> LoadPackage("vole", false);;
```

Example

```
gap> ret := Vole.Intersection(AlternatingGroup(6), PGL(2,5) : raw := true);
rec( cosetrep := (), group := Group([ (3,5)(4,6), (2,3,4,6,5), (1,2,6)(3,4,5)
]),
  raw := rec( canonical := fail, rbase_branches := [ 1, 2, 3 ],
    search_fix_order := [ 1, 2, 3, 6, 5, 4 ],
    sols := [ [ ], [ 1, 2, 5, 6, 3, 4 ], [ 1, 3, 4, 6, 2, 5 ],
      [ 2, 6, 4, 5, 3, 1 ] ],
    stats := rec( bad_canonical := 0, bad_iso := 3, equal_canonical := 0,
      gap_callbacks := rec( begin := 2, canonicalmin_time := 0,
        changed := 24, check := 11, compare := 0, fixed := 38,
        image := 0, is_group := 0, name := 0, rBaseFinished := 2,
        refiner_time := 40, restore_state := 24, save_state := 24 ),
```

```

        good_iso := 4, improve_canonical := 0, refiner_calls := 64,
        search_nodes := 13, trace_fail_nodes := 0, vole_time := 136 ) ),
    sols := [ (), (3,5)(4,6), (2,3,4,6,5), (1,2,6)(3,4,5) ], time := 220 )
gap> ret.group = PSL(2,5);
true

```

Note that the option `raw` can also be set to `true` by simply giving the name `raw`. Therefore the following line can be used in place of the one above:

Example

```

gap> ret := Vole.Intersection(AlternatingGroup(6), PGL(2,5) : raw);;

```

8.2.1 Supported options

- `raw`: true or (default) false:
 - If true, then rather than returning an object as documented, the function returns a record that contains detailed information about the execution and result of the graph backtracking algorithm that was executed. The meaning of this information is not yet documented, sorry.
 - Not all *Vole* functions support this option, although most do.
- `points`: a non-negative integer or (default) infinity:
 - This option can be used in place of the constraint `Constraint.LargestMovedPoint(points)`; see `Constraint.LargestMovedPoint` (6.4.11).
 - Note that for relevant functions that accept constraints, an integer argument is interpreted as an instance of a ‘largest moved point’ constraint, which achieves the same affect as this option more simply.

Example

```

gap> D := PetersenGraph();;
gap> constraint := Constraint.Stabilise(D, OnDigraphs);;
gap> G := VoleFind.Group(constraint : points := DigraphNrVertices(D));;
gap> [NrMovedPoints(G), TransitiveIdentification(G)];
[ 10, 13 ]

```

References

- [BJM⁺21] Jan De Beule, Julius Jonušas, James D. Mitchell, Michael Torpey, Maria Tsalakou, and Wilf A. Wilson. *Digraphs – GAP package, Version 1.5.0*, 10 2021. [6](#), [9](#)
- [GAP21] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.11.1*, March 2021. [5](#), [6](#)
- [Jef20] Christopher Jefferson. *json – GAP package, Version 2.0.2*, 4 2020. [10](#)
- [Jef21] Christopher Jefferson. *ferret – GAP package, Version 1.0.6*, 10 2021. [6](#), [10](#)
- [JHPL21] Christopher Jefferson, Max Horn, Markus Pfeiffer, and Steve Linton. *datastructures – GAP package, Version 0.2.6*, 4 2021. [9](#)
- [JJPW19] Christopher Jefferson, Eliza Jonauskyte, Markus Pfeiffer, and Rebecca Waldecker. Minimal and canonical images. *J. Algebra*, 521:481–506, 2019. [5](#), [6](#)
- [JJPW21] Christopher Jefferson, Eliza Jonauskyte, Markus Pfeiffer, and Rebecca Waldecker. *images – GAP package, Version 1.3.1*, 11 2021. [6](#)
- [JK07] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 135–149, Jan 2007. [6](#)
- [JPW19] Christopher Jefferson, Markus Pfeiffer, and Rebecca Waldecker. New refiners for permutation group search. *J. Symbolic Comput.*, 92:70–92, 2019. [6](#)
- [JPWW19] Christopher Jefferson, Markus Pfeiffer, Rebecca Waldecker, and Wilf A. Wilson. Permutation group algorithms based on directed graphs (extended version). *manuscript*, 2019. [5](#)
- [JPWW21] Christopher Jefferson, Markus Pfeiffer, Rebecca Waldecker, and Wilf A. Wilson. Permutation group algorithms based on directed graphs. *J. Algebra*, 585:723–758, 2021. [5](#), [7](#)
- [JW21a] Christopher Jefferson and Wilf A. Wilson. *BacktrackKit – GAP package, Version 0.6.0*, 12 2021. [6](#)
- [JW21b] Christopher Jefferson and Wilf A. Wilson. *GraphBacktracking – GAP package, Version 0.5.1*, 12 2021. [6](#)
- [Leo91] Jeffrey S. Leon. Permutation group algorithms based on partitions. I. Theory and algorithms. *J. Symbolic Comput.*, 12(4-5):533–583, 1991. [5](#), [6](#)

- [Leo97] Jeffrey S. Leon. Partitions, refinements, and permutation group computation. In *Groups and computation, II (New Brunswick, NJ, 1995)*, volume 28 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pages 123–158. Amer. Math. Soc., Providence, RI, 1997. [5](#), [6](#)
- [Lin04] Steve Linton. Finding the smallest image of a set. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, pages 229–234, 2004. [6](#)
- [MP14] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symbolic Comput.*, 60:94–112, 2014. [6](#)
- [Neu21] Max Neunhöffer. *IO – GAP package, Version 4.7.2*, 10 2021. [9](#)
- [Soi21] L. H. Soicher. *GRAPE – GAP package, Version 4.8.5*, 3 2021. [6](#)
- [The97] Heiko Theißen. *Eine Methode zur Normalisatorberechnung in Permutationsgruppen mit Anwendungen in der Konstruktion primitiver Gruppen*. Verlag der Augustinus-Buchh., 1997. [6](#)

Index

Constraint, [37](#)
Constraint.Centralise, [41](#)
Constraint.Centralize, [41](#)
Constraint.Conjugate, [41](#)
Constraint.Everything, [43](#)
Constraint.InCoset, [38](#)
Constraint.InGroup, [38](#)
Constraint.InLeftCoset, [39](#)
Constraint.InRightCoset, [39](#)
Constraint.IsEven, [42](#)
Constraint.IsOdd, [42](#)
Constraint.IsTrivial, [42](#)
Constraint.LargestMovedPoint, [42](#)
Constraint.MovedPoints, [41](#)
Constraint.None, [43](#)
Constraint.Normalise, [41](#)
Constraint.Normalize, [41](#)
Constraint.Stabilise, [40](#)
Constraint.Stabilize, [40](#)
Constraint.Transport, [39](#)

InfoVole, [47](#)

Vole, [19](#)
Vole.AutomorphismGroup, [26](#)
Vole.CanonicalDigraph, [27](#)
Vole.CanonicalImage, [25](#)
Vole.CanonicalImagePerm, [24](#)
Vole.CanonicalPerm, [24](#)
Vole.Centraliser, [22](#)
Vole.Centralizer, [22](#)
Vole.DigraphCanonicalLabelling, [27](#)
Vole.Intersection, [20](#)
Vole.IsConjugate, [23](#)
Vole.IsIsomorphicDigraph, [27](#)
Vole.IsomorphismDigraphs, [28](#)
Vole.Normaliser, [22](#)
Vole.Normalizer, [22](#)
Vole.RepresentativeAction, [21](#)

Vole.Stabiliser, [20](#)
Vole.Stabilizer, [20](#)
Vole.TwoClosure, [23](#)
VoleFind, [29](#)
VoleFind.Canonical, [32](#)
VoleFind.CanonicalPerm, [35](#)
VoleFind.Coset, [31](#)
VoleFind.Group, [30](#)
VoleFind.Rep, [29](#)
VoleFind.Representative, [29](#)
VoleRefiner, [44](#)
VoleRefiner.DigraphStab, [45](#)
VoleRefiner.DigraphTransporter, [45](#)
VoleRefiner.FromConstraint, [46](#)
VoleRefiner.InSymmetricGroup, [44](#)
VoleRefiner.SetSetStab, [45](#)
VoleRefiner.SetSetTransporter, [45](#)
VoleRefiner.SetStab, [45](#)
VoleRefiner.SetTransporter, [45](#)
VoleRefiner.SetTupleStab, [45](#)
VoleRefiner.SetTupleTransporter, [45](#)
VoleRefiner.TupleStab, [45](#)
VoleRefiner.TupleTransporter, [45](#)