

GraphBacktracking

**A simple but slow implementation of
graph backtracking**

1.2.0

1 October 2026

Christopher Jefferson
Wilf A. Wilson

Christopher Jefferson Email: caj21@st-andrews.ac.uk
Homepage: <http://caj.host.cs.st-andrews.ac.uk/>
Address: St Andrews
Scotland
UK

Wilf A. Wilson Email: gap@wilf-wilson.net
Homepage: <https://wilf.me>

Contents

1	Introduction	3
1.1	What is GraphBacktracking?	3
1.2	Relationship to BacktrackKit and Vole	3
2	Executing a search	4
2.1	Information and diagnostics	4
2.2	The main search interface	4
2.3	Inspecting the initial graph stack	5
3	Refiners	6
3.1	The GB Con record	6
3.2	Group and coset refiners	6
3.3	Transporter refiners	7
3.4	Normaliser and group-conjugacy refiners	7
3.5	Experimental normaliser variants	7
4	Equitable Graphs	9
4.1	Making a partition equitable	9
	Index	11

Chapter 1

Introduction

1.1 What is GraphBacktracking?

GraphBacktracking implements the graph backtracking algorithm described in the paper “Computing canonical images in permutation groups with Graph Backtracking”, by Christopher Jefferson, Rebecca Waldecker and Wilf A. Wilson (<https://arxiv.org/abs/2209.02534>). Graph backtracking generalises Leon’s partition backtrack: instead of refining only an ordered partition of the points, the search also accumulates a stack of vertex- and edge-labelled graphs, which can capture constraints that a partition alone cannot. This makes it possible to solve problems such as normaliser computation and the canonical image of a graph under an arbitrary group.

This package builds directly on the **BacktrackKit** package, and reuses its framework essentially unchanged: ordered partition stacks, tracers, constraints, the refiner protocol, and the top-level search loop. Those concepts are documented in the **BacktrackKit** manual and are not repeated here. This manual assumes you are familiar with them, in particular with (**BacktrackKit: The concept of constraints**) and (**BacktrackKit: Refiners**), and documents only what GraphBacktracking adds on top.

1.2 Relationship to BacktrackKit and Vole

Like **BacktrackKit**, this package exists for *learning and exploring* the algorithms. Its performance is *extremely poor* — often orders of magnitude slower than the built-in **GAP** functions for the same task. For a modern, high-performance implementation of graph backtracking, use the **vole** package (<https://github.com/peal/vole>) instead.

The one substantive extension to the **BacktrackKit** refiner protocol is that a GraphBacktracking refiner may, in addition to splitting cells of the partition, emit *graphs*. A value returned by a refiner’s `refine` functions (see (**BacktrackKit: The record refine**)) may be a record with the components:

- `graph` — a **Digraphs** digraph that is added to the graph stack; and/or
- `vertlabels` — a list giving an initial colour for each vertex.

The graph stack is then made equitable (see 4) and contributes to the branching and pruning of the search in the same way the partition does.

Chapter 2

Executing a search

2.1 Information and diagnostics

GraphBacktracking reports diagnostic information through the info class `InfoGB`, which is set equal to the `BacktrackKit` info class `InfoBTKit` (**BacktrackKit: InfoBTKit**); raising the level of either (with `SetInfoLevel`) therefore raises both. Higher levels print progressively more detail about the progress of the search.

2.2 The main search interface

These three functions are the graph backtracking analogues of the `BacktrackKit` search functions `BTKit_SimpleSearch` (**BacktrackKit: BTKit_SimpleSearch**), `BTKit_SimpleSinglePermSearch` (**BacktrackKit: BTKit_SimpleSinglePermSearch**) and `BTKit_SimpleAllPermSearch` (**BacktrackKit: BTKit_SimpleAllPermSearch**), and take the same arguments. Each begins from the ordered partition stack *ps* (see `PartitionStack` (**BacktrackKit: PartitionStack**)) and searches for the permutations that satisfy every refiner in the list *conlist*. Refiners may be built with the constructors in the `GB_Con` record (see 3) or with any `BacktrackKit` refiner. The optional final argument *conf* is a configuration record; the recognised fields are `cellSelector` (the branch-cell selector) and `consolidator` (the function used to make the graph stack equitable, see 4).

2.2.1 GB_SimpleSearch

▷ `GB_SimpleSearch(ps, conlist[, conf])` (function)
Returns: a permutation group

Returns the group of all permutations that satisfy every refiner in *conlist*. This is correct only when that set of permutations is in fact a group (for example, an intersection of groups, a stabiliser, or a normaliser); use `GB_SimpleSinglePermSearch` (2.2.2) when the solution set is a coset.

2.2.2 GB_SimpleSinglePermSearch

▷ `GB_SimpleSinglePermSearch(ps, conlist[, conf])` (function)
Returns: a permutation, or fail

Returns a single permutation satisfying every refiner in *conlist*, or fail if no such permutation exists. This is the function to use for coset and transporter problems (group or coset intersection, conjugacy, and so on).

2.2.3 GB_SimpleAllPermSearch

▷ `GB_SimpleAllPermSearch(ps, conlist[, conf])` (function)

Returns: a list of permutations

Returns the complete list of permutations satisfying every refiner in *conlist*. This enumerates the whole solution set and is therefore very slow; it is intended for testing and exploration on small examples.

2.3 Inspecting the initial graph stack

These functions expose the graph stack that is built before any branching takes place. They are used mainly for testing and for understanding how much a set of refiners deduces “for free”, and are unlikely to be needed in ordinary use.

2.3.1 GB_CheckInitialGroup

▷ `GB_CheckInitialGroup(ps, conlist)` (function)

Returns: a record

Builds the initial graph stack for *conlist* and returns a record with components *gens* (generators of the automorphism group of the initial stack, as permutations) and *answer* (true if every one of those generators already satisfies *conlist*, so that no search is required; false otherwise, in which case the group is a supergroup of the solutions).

2.3.2 GB_CheckInitialCoset

▷ `GB_CheckInitialCoset(ps, conlist)` (function)

Returns: a record

The coset analogue of `GB_CheckInitialGroup` (2.3.1). Builds the initial graph stack down both the left and the right branch and returns a record with components *graph1*, *graph2* (the two canonically-labelled graph stacks) and *equal* (whether they coincide).

Chapter 3

Refiners

3.1 The GB_Con record

GraphBacktracking provides its refiners through the global record `GB_Con`, in the same spirit as the `BTKit_Refiner` record of `BacktrackKit`. Each field of `GB_Con` is a function that constructs a refiner; the resulting refiner can be placed in the `conlist` argument of the search functions in 2, freely mixed with `BacktrackKit` refiners. For the general notion of a refiner and the constraint it refines for, see **(BacktrackKit: Refiners)** and **(BacktrackKit: Constraints)**. What is specific to this package is that these refiners may additionally push graphs onto the graph stack, as described in 1.

The descriptions below cover the refiners intended for ordinary use. Several of them have a family of experimental variants (used for benchmarking the different graph constructions); those are listed in 3.5.

3.2 Group and coset refiners

`GB_Con.InGroup( $G$ )`

refines for the permutations lying in the group G . Placing two such refiners in a search computes a group intersection.

`GB_Con.InCoset( $G, x$ )`

refines for the permutations lying in the right coset $G * x$.

`GB_Con.InGroupSimple` and `GB_Con.InCosetSimple` are alternative implementations of the same two constraints that push a different (simpler) set of graphs; they exist for comparison and compute the same answers.

Example

```
gap> LoadPackage("graphbacktracking", false);;
gap> G := SymmetricGroup(4);;
gap> H := DihedralGroup(IsPermGroup, 8);;
gap> GB_SimpleSearch(PartitionStack(4),
> [GB_Con.InGroup(G), GB_Con.InGroup(H)]) = Intersection(G, H);
true
```

3.3 Transporter refiners

`GB_Con.PermConjugacy(a, b)`

refines for the permutations p with $a \sim p = b$, i.e. that conjugate the permutation a to b .

`GB_Con.TransformationConjugacy(a, b)`

likewise for transformations a, b : refines for the permutations p with $a \sim p = b$ (conjugation), via the functional digraph of each transformation.

`GB_Con.PartialPermConjugacy(a, b)`

likewise for partial permutations a, b .

`GB_Con.SetDigraphs(setL, setR)`

refines for the permutations mapping the set of digraphs $setL$ to the set of digraphs $setR$ (under `OnSetsDigraphs`). With $setL = setR$ this is the setwise stabiliser of a set of digraphs.

3.4 Normaliser and group-conjugacy refiners

These are the refiners that motivate graph backtracking, and the main addition in this release. To compute the normaliser of G inside some group U , combine the normaliser refiner with a `BTKit_Refiner.InGroup(U)` (or `GB_Con.InGroup`) refiner.

`GB_Con.NormaliserOrbital(G)`

the recommended refiner for the normaliser of G . It pushes the orbital graphs of the relevant stabilisers (encoded as a set of graphs the normaliser may permute), together with orbit colourings and root-level block systems.

`GB_Con.GroupConjugacyOrbital(L, R)`

the transporter version: refines for the permutations conjugating the group L to the group R (group conjugacy under `OnPoints`). `GB_Con.NormaliserOrbital(G)` is exactly `GB_Con.GroupConjugacyOrbital(G, G)`.

Example

```
gap> G := Group((1,2,3,4,5));;
gap> N := GB_SimpleSearch(PartitionStack(5),
> [BTKit_Refiner.InGroup(SymmetricGroup(5)),
> GB_Con.NormaliserOrbital(G)]);;
gap> N = Normaliser(SymmetricGroup(5), G);
true
```

3.5 Experimental normaliser variants

For experimenting with the different graph constructions, several other normaliser refiners are provided. They all compute the same normaliser as `GB_Con.NormaliserOrbital` (and the corresponding `GroupConjugacy*` two-argument forms exist as well), but differ in which graphs and deductions they push, and hence in speed: `NormaliserSimple`, `NormaliserSimple2`, `NormaliserOrbitalRoot`, `NormaliserOrbitalNone`, `NormaliserOrbitalDeep`, `NormaliserOrbitalSmall`, `NormaliserOrbitalRegOrbit` and

NormaliserOrbitalRegOrbitChar. The trade-offs of each are documented in the comments of `gap/constraints/normaliser.g`.

WARNING. The two regular-orbit variants (NormaliserOrbitalRegOrbit and NormaliserOrbitalRegOrbitChar) are *not* canonical-safe: they are correct for testing normaliser/conjugacy equality, but must not be used for canonical-image computations. Use `GB_Con.NormaliserOrbital` when a canonical image is required.

Chapter 4

Equitable Graphs

4.1 Making a partition equitable

During search, the partition must be refined with respect to the current graph stack until it is *equitable*: informally, until no cell can be split by looking at how its points connect to the cells of the partition. GraphBacktracking offers several methods of differing strength and cost. A stronger method splits the partition at least as much as a weaker one, prunes the search tree more, but costs more per node. Which one is used is controlled by the `consolidator` field of the configuration record passed to the search functions (see 2); the default is `GB_MakeEquitableStrong` (4.1.3).

Each method takes a partition stack *ps*, a tracer *tracer* (see **(BacktrackKit: Ordered tracers)**), and a list *graphs* of digraphs. It refines *ps* in place, recording the splits in *tracer*, and returns `true` on success or `false` if a split contradicted the tracer (a dead branch).

4.1.1 GB_MakeEquitableNone (for IsPartitionStack, IsTracer, IsList)

▷ `GB_MakeEquitableNone(ps, tracer, graphs)` (operation)

Does nothing and returns `true`: the partition is left unchanged. Provided as a baseline (it makes graph backtracking behave like ordinary backtracking with respect to the graphs).

4.1.2 GB_MakeEquitableWeak (for IsPartitionStack, IsTracer, IsList)

▷ `GB_MakeEquitableWeak(ps, tracer, graphs)` (operation)

Refines *ps* by repeatedly splitting each cell according to the multiset of cells reached along out- and in-edges of each graph, iterating to a fixed point. This is the classical equitable-partition refinement applied to each graph in turn.

4.1.3 GB_MakeEquitableStrong (for IsPartitionStack, IsTracer, IsList)

▷ `GB_MakeEquitableStrong(ps, tracer, graphs)` (operation)

A stronger refinement than `GB_MakeEquitableWeak` (4.1.2): it distinguishes points using the combined edge information across all graphs simultaneously, rather than one graph at a time. This is the default consolidator.

4.1.4 GB_MakeEquitableFull (for IsPartitionStack, IsTracer, IsList)

▷ `GB_MakeEquitableFull(ps, tracer, graphs)` (operation)

The strongest (and most expensive) refinement: it builds a single digraph encoding the whole graph stack, computes its automorphism group and orbits, and splits the partition by those orbits. This can prune branches the cheaper methods miss, at a substantial cost per node.

Index

GB_CheckInitialCoset, [5](#)
GB_CheckInitialGroup, [5](#)
GB_MakeEquitableFull
 for IsPartitionStack, IsTracer, IsList, [10](#)
GB_MakeEquitableNone
 for IsPartitionStack, IsTracer, IsList, [9](#)
GB_MakeEquitableStrong
 for IsPartitionStack, IsTracer, IsList, [9](#)
GB_MakeEquitableWeak
 for IsPartitionStack, IsTracer, IsList, [9](#)
GB_SimpleAllPermSearch, [5](#)
GB_SimpleSearch, [4](#)
GB_SimpleSinglePermSearch, [4](#)