

BacktrackKit

**An extensible, easy to understand
backtracking framework**

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: School of Computer Science

University of St Andrews

Jack Cole Building, North Haugh

St Andrews, Fife, KY16 9SX

United Kingdom

Wilf A. Wilson Email: gap@wilf-wilson.net

Homepage: <https://wilf.me>

Contents

1	Introduction	3
2	Executing a search	4
2.1	Extra information and stats	4
2.2	The main search interface	4
3	Implementation	5
3.1	State	5
3.2	Applying refiners	6
3.3	Currently unorganised stuff	7
4	Constraints	8
4.1	The concept of constraints	8
4.2	The Constraints record	9
4.3	Constraints via the Constraint record	9
5	Refiners	16
5.1	Introduction to refiners	16
5.2	The record refine	16
5.3	Constructing refiners	17
6	Ordered partition stacks	18
6.1	API	18
7	Ordered tracers	22
7.1	API	22
8	Tutorial	24
8.1	Partition Stacks	24
8.2	Refiners	25
	References	27
	Index	28

Chapter 1

Introduction

BacktrackKit is a package which does some interesting and cool things.

This implements the algorithm described in [\[Leo91\]](#) and [\[Leo97\]](#).

Chapter 2

Executing a search

2.1 Extra information and stats

2.1.1 InfoBTKit

▷ InfoBTKit (info class)

Information about backtrack search.

2.1.2 BTKit_ResetStats

▷ BTKit_ResetStats() (function)

2.2 The main search interface

Search for generators of a group

2.2.1 BTKit_SimpleSearch

▷ BTKit_SimpleSearch(*arg*) (function)

Search for a single permutation

2.2.2 BTKit_SimpleSinglePermSearch

▷ BTKit_SimpleSinglePermSearch(*arg*) (function)

Search for all permutations (very slow)

2.2.3 BTKit_SimpleAllPermSearch

▷ BTKit_SimpleAllPermSearch(*arg*) (function)

Chapter 3

Implementation

3.1 State

3.1.1 IsBacktrackableState (for IsObject)

▷ IsBacktrackableState(*arg*) (Category)
Returns: true or false

3.1.2 IsBTKitState (for IsBacktrackableState)

▷ IsBTKitState(*arg*) (Representation)
Returns: true or false

3.1.3 BacktrackableStateFamily

▷ BacktrackableStateFamily (global variable)

3.1.4 BTKitStateType

▷ BTKitStateType (global variable)

3.1.5 SaveState (for IsBacktrackableState)

▷ SaveState(*arg*) (operation)
Returns: The saved state
Return a small object which allows one to revert to this state from later the search.

3.1.6 RestoreState (for IsBacktrackableState, IsObject)

▷ RestoreState(*state*, *saved*) (operation)
Returns: Nothing
Revert to a saved state from later in the search. The first argument *state* must be the current state object, and the second argument *saved* must be one of the objects produced by SaveState (3.1.5) from earlier in the search.

3.1.7 ConsolidateState (for IsBacktrackableState, IsTracer)

▷ ConsolidateState(*arg1*, *arg2*) (operation)

Some implementations of `BacktrackableState` can perform simplifications. This function gives a well-defined point for such operations to be performed. It can be ignored by implementations without such simplifications.

3.2 Applying refiners

3.2.1 InitialiseConstraints

▷ InitialiseConstraints(*state*, *tracer*, *rbase*) (function)

Returns: true or false

Set up the list of constraints in `state.conlist`, using their `refine.initialise` members. This should be called once at the start of RBase creation, and once at the start of search. During search, if the branch of search becomes inconsistent with the RBase, then this function returns false. Otherwise, this function returns true.

The second and third arguments *tracer* and *rbase* should be as in `RefineConstraints` (3.2.2).

3.2.2 RefineConstraints

▷ RefineConstraints(*state*, *tracer*, *rbase*) (function)

Returns: true or false

Refine the partition stack `state.ps` according to the list of constraints in `state.conlist`, until it is not possible to use them to refine the current partition stack any further, or until the branch of search becomes inconsistent with the RBase. In the former case, this function returns true, and in the latter case, this function returns false.

During RBase creation, the second argument *tracer* must be a recording tracer, and the third argument *rbase* must be true. During search, the second argument should be a tracer following the corresponding RBase tracer, and the third argument *rbase* should be false.

3.2.3 FinaliseRBaseForConstraints

▷ FinaliseRBaseForConstraints(*arg*) (function)

Set up a list of constraints. This should be called once, at the start of search after all constraints have been created.

3.2.4 ApplyFilters (for IsBTKitState, IsTracer, IsObject)

▷ ApplyFilters(*ps*, *tracer*, *filter*) (operation)

Returns: true or false

Split the cells of the partition stack *ps*, if possible, according to a given *filter*. If the filter is fail, or if the split is rejected by the *tracer*, then this function returns false. Otherwise, the split is applied and is consistent with the *tracer*, and this function returns true.

3.3 Currently unorganised stuff

3.3.1 BTKit_BuildProblem

▷ BTKit_BuildProblem(*arg*) (function)

Takes a partition stack and a list of constraints and builds a 'Problem', Which can then be solved by passing the 'Problem' to BTKit_SimpleSearch (2.2.1) or BTKit_SimpleSinglePermSearch (2.2.2).

3.3.2 FirstFixedPoint

▷ FirstFixedPoint(*arg*) (function)

3.3.3 BuildRBase

▷ BuildRBase(*arg*) (function)

3.3.4 Backtrack

▷ Backtrack(*arg*) (function)

Chapter 4

Constraints

4.1 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` (4.2.1) record. The members of this record are documented individually in Section 4.3.

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 5, 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.

4.2 The Constraints record

4.2.1 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 4.3.

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

Example

```
gap> for c in Set(RecNames(Constraint)) do Print(c, "\n"); od;
Centralise
Centralize
Conjugate
Everything
InCoset
InGroup
InLeftCoset
InRightCoset
IsEven
IsOdd
IsTrivial
LargestMovedPoint
MovedPoints
None
Normalise
Normalize
Nothing
Stabilise
Stabilize
Transport
```

4.3 Constraints via the Constraint record

In this section, we individually document the functions of the Constraint (4.2.1) 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> (4.3.1)	<code>Constraint.InCoset</code> (4.3.2) <code>Constraint.InRightCoset</code> (4.3.3) <code>Constraint.InLeftCoset</code> (4.3.4)
<code>Constraint.Stabilise</code> (4.3.6)	<code>Constraint.Transport</code> (4.3.5)
<code>Constraint.Normalise</code> (4.3.7)	<code>Constraint.Conjugate</code> (4.3.9)
<code>Constraint.Centralise</code> (4.3.8)	<code>Constraint.Conjugate</code> (4.3.9)
<code>Constraint.MovedPoints</code> (4.3.10)	N/A
<code>Constraint.LargestMovedPoint</code> (4.3.11)	N/A
<code>Constraint.IsEven</code> (4.3.12)	<code>Constraint.IsOdd</code> (4.3.13)
<code>Constraint.IsTrivial</code> (4.3.14)	N/A
N/A	<code>Constraint.None</code> (4.3.15)

4.3.1 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 ] )>
```

4.3.2 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` (4.3.4) and `Constraint.InRightCoset` (4.3.3), 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)>
```

4.3.3 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` (4.3.4) for the left-hand version, and `Constraint.InCoset` (4.3.2) 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)>
```

4.3.4 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` (4.3.3) for the right-hand version, and `Constraint.InCoset` (4.3.2) 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)
```

4.3.5 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.

If you do not give the optional *action* argument, then it defaults to `OnPoints`. This default suits a point, a permutation, or a permutation group; for the other kinds of object below you must name the action yourself.

Object	Action
A point (a positive integer)	<code>OnPoints</code>
A list of points	<code>OnTuples</code>
A set of points	<code>OnSets</code>
A permutation	<code>OnPoints</code>
A permutation group	<code>OnPoints</code>
A digraph (from the <code>Digraphs</code> package)	<code>OnDigraphs</code>

BacktrackKit solves each combination in this table with a dedicated refiner. Any other object and action still gives a correct answer, but BacktrackKit has no specialised refiner for it, and so falls back to an unrefined search that tests each candidate permutation directly. This is correct, but it can be very slow.

Example

```
gap> set1 := [1, 3, 6];;
gap> set2 := [2, 4, 5];;
gap> con := Constraint.Transport(set1, set2, OnSets);
<constraint: transporter of [ 1, 3, 6 ] to [ 2, 4, 5 ] under OnSets>
```

4.3.6 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.

If you do not give the optional *action* argument, then it defaults to `OnPoints`. This default suits a point, a permutation, or a permutation group; for the other kinds of object below you must name the action yourself.

Object	Action
A point (a positive integer)	<code>OnPoints</code>
A list of points	<code>OnTuples</code>
A set of points	<code>OnSets</code>
A permutation	<code>OnPoints</code>
A permutation group	<code>OnPoints</code>
A digraph (from the <code>Digraphs</code> package)	<code>OnDigraphs</code>

BacktrackKit solves each combination in this table with a dedicated refiner. Any other object and action still gives a correct answer, but BacktrackKit has no specialised refiner for it, and so falls back to an unrefined search that tests each candidate permutation directly. This is correct, but it can be very slow.

Example

```
gap> con1 := Constraint.Stabilise(CycleDigraph(6), OnDigraphs);
<constraint: stabiliser of <immutable cycle digraph with 6 vertices> under OnD\
igraphs>
gap> con2 := Constraint.Stabilise([2,4,6], OnSets);
<constraint: stabiliser of [ 2, 4, 6 ] under OnSets>
```

4.3.7 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) ] )>
```

4.3.8 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)>
```

4.3.9 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)>
```

4.3.10 Constraint.MovedPoints

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

Returns: A constraint

This constraint is a shorthand for `Constraint.InGroup(SymmetricGroup(<A>pointlist</A>))`. See `Constraint.InGroup` (4.3.1).

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 ]>
```

4.3.11 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` (4.3.1).

Example

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

4.3.12 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);
()
```

4.3.13 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)
```

4.3.14 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);
()
```

4.3.15 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
```

4.3.16 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 5

Refiners

5.1 Introduction to refiners

In `BacktrackKit`, a refiner is implemented as a record that must contain:

- A member called `name`, which is a string giving the name of the constraint;
- A member called `largest_required_point`, which is an integer which gives the smallest size partition this refiner will work on. For example, given a set we would expect this to be the largest element of the set.
- A member called `constraint`, which is a constraint object, such that the refiner is refining for the permutations that satisfy the constraint.
- A member called `refine`, which is a record; more information is given below.

A constraint may also optionally contain any of the following members:

- A member called `btdata`. The data in this member will be automatically saved and restored on backtrack.

5.2 The record `refine`

The `refine` member of a constraint is a record that contains functions which, if present, will be called to inform the constraint of behaviour as search progresses, and to give the constraint the opportunity to influence the search. The permissible functions are given described below.

These functions will always be passed at least two arguments: firstly the constraint itself, and then the partition stack. Details of any further arguments are described with the relevant function, below.

- `initialise` (*required*). This is called when search begins. Note that, since the `refine.initialise` function is called for all relevant constraints at the beginning of search, the partition may have already been split by some earlier constraint by the time that `refine.initialise` is called for a later constraint.

At most one of the following two functions will generally be implemented.

- `changed` - Will be called after one or more splits in the partition occur.
- `fixed` - Will be called after one or more points in the partition became fixed.
- `rBaseFinished` - The `rBase` has been created -- this is passed the partition which was generated down the first branch of search. Constraints which care about this can use this to remember the `rBase` construction is finished.
- `solutionFound` - A solution to the problem has been found. This function is passed a permutation. This function is rarely needed.

5.3 Constructing refiners

5.3.1 DummyRefiner (for IsConstraint)

▷ `DummyRefiner(constraint)`

(attribute)

This wraps a constraint in a refiner that doesn't do any refining.

Chapter 6

Ordered partition stacks

An *ordered partition stack* is an ordered partition which supports splitting a cell, and then later undoing a change, reverting the partition back to an earlier state.

6.1 API

6.1.1 PartitionStack

- ▷ PartitionStack() (function)
Returns: A partition stack
Constructor for partition stacks.

6.1.2 IsPartitionStack (for IsObject)

- ▷ IsPartitionStack(arg) (Category)
Returns: true or false
Category of partition stacks.

6.1.3 PS_Points (for IsPartitionStack)

- ▷ PS_Points(*PS*) (operation)
Returns: A positive integer
Return the number of points on which the partition stack *PS* was originally defined.

6.1.4 PS_ExtendedPoints (for IsPartitionStack)

- ▷ PS_ExtendedPoints(*PS*) (operation)
Returns: A positive integer
Return the number of points on which the partition stack *PS* is currently defined on (which includes extra points added during refinement)

6.1.5 PS_Extend (for IsPartitionStack, IsPosInt)

- ▷ PS_Extend(*PS*, *NewPoints*) (operation)
Returns: The label of the new cell
Adds *NewPoints* to *PS*, as a new cell at the end. Will be removed on backtracking.

6.1.6 PS_Cells (for IsPartitionStack)

▷ `PS_Cells(PS)` (operation)

Returns: A positive integer

The number of cells in the current partition state of the partition stack *PS*.

6.1.7 PS_Fixed (for IsPartitionStack)

▷ `PS_Fixed(PS)` (operation)

Returns: true or false

Return true if all the cells of the current partition state of the partition stack *PS* have size 1 and were in the 'original partition'.

6.1.8 PS_AsPartition (for IsPartitionStack)

▷ `PS_AsPartition(PS)` (operation)

Returns: A list of lists of positive integers

Return the current partition state of the partition stack *PS* as a list of sets, in the correct order.

6.1.9 PS_CellLen (for IsPartitionStack, IsPosInt)

▷ `PS_CellLen(PS, i)` (operation)

Returns: A positive integer

The size of cell *i* in the current partition state of the partition stack *PS*. This requires that *i* is contained in `[1..PS_Cells(PS)]`.

6.1.10 PS_CellSlice (for IsPartitionStack, IsPosInt)

▷ `PS_CellSlice(PS, i)` (operation)

Returns: An immutable list of positive integers

Return an immutable list containing the elements of cell *i* in the current partition state of the partition stack *PS*. This requires that *i* is contained in `[1..PS_Cells(PS)]`.

6.1.11 PS_FixedCells (for IsPartitionStack)

▷ `PS_FixedCells(PS)` (operation)

Returns: A list of 1-element lists of positive integers

Return a list of the 1-element cells of the current state of the partition stack *PS* which contain points from the original size of the partition, in the order in which the cells came to have size 1.

6.1.12 PS_FixedPoints (for IsPartitionStack)

▷ `PS_FixedPoints(PS)` (operation)

Returns: A list of positive integers

Return a list of points in 1-element cells of the partition stack *PS*, in the order in which the cells came to have size 1. In other words, return `Concatenation(PS_FixedCells(PS))`.

6.1.13 PS_CellOfPoint (for IsPartitionStack, IsPosInt)

▷ `PS_CellOfPoint(PS, i)` (operation)

Returns: A positive integer

Return the index of the cell containing the value *i* in the current partition state of the partition stack *PS*. This requires that *i* is contained in `[1..PS_ExtendedPoints(PS)]`.

6.1.14 PS_RevertToCellCount (for IsPartitionStack, IsPosInt)

▷ `PS_RevertToCellCount(PS, i)` (operation)

Returns: Nothing

Revert the state of the partition stack *PS* to when there were *i* cells. This requires that *i* is contained in `[1..PS_Cells(PS)]`.

6.1.15 PS_SplitCellByFunction (for IsPartitionStack, IsTracer, IsPosInt, IsFunction)

▷ `PS_SplitCellByFunction(PS, t, i, f)` (operation)

Returns: true or false

Split cell *i* of the current partition state of the partition stack *PS* according to the function *f*. The values in the cell are split so that those with different images under *f* are put into different cells. The second argument *t* should be a tracer.

6.1.16 PS_SplitCellsByFunction (for IsPartitionStack, IsTracer, IsFunction)

▷ `PS_SplitCellsByFunction(PS, t, f)` (operation)

Returns: true or false

Apply `PS_SplitCellByFunction` to every active cell in the partition stack *PS* (ignoring points added after search starts).

6.1.17 PS_ExtendedSplitCellsByFunction (for IsPartitionStack, IsTracer, IsFunction)

▷ `PS_ExtendedSplitCellsByFunction(PS, t, f)` (operation)

Returns: true or false

Apply `PS_SplitCellByFunction` to every active cell in the partition stack *PS* (including points added after search starts).

6.1.18 PS_SplitCellByUnorderedFunction (for IsPartitionStack, IsTracer, IsPosInt, IsFunction)

▷ `PS_SplitCellByUnorderedFunction(PS, t, i, f)` (operation)

Split cell *i* of the current partition state of the partition stack *PS* according to the function *f*. The values in the cell are split so that those with different images under *f* are put into different cells. The second argument *t* should be a tracer.

6.1.19 PS_SplitCellsByUnorderedFunction (for IsPartitionStack, IsTracer, IsFunction)

▷ `PS_SplitCellsByUnorderedFunction(PS, t, f)` (operation)

Apply `PS_SplitCellByUnorderedFunction` to every active cell in the partition stack *PS*. The second argument *t* should be a tracer.

Chapter 7

Ordered tracers

An *ordered tracer* is...

7.1 API

7.1.1 RecordingTracer

- ▷ `RecordingTracer()` (function)
Returns: A recording tracer
Constructor for recording tracers.

7.1.2 FollowingTracer

- ▷ `FollowingTracer(t)` (function)
Returns: A following tracer
If the argument *t* is a tracer, then this constructs a tracer that follows *t*.

7.1.3 CanonicalisingTracerFromTracer

- ▷ `CanonicalisingTracerFromTracer()` (function)
Returns: A canonicalising tracer
Constructor for canonicalising tracers.

7.1.4 IsTracer (for IsObject)

- ▷ `IsTracer(t)` (Category)
Returns: true or false
The category of tracers.

7.1.5 AddEvent (for IsTracer, IsObject)

- ▷ `AddEvent(t, o)` (operation)
Returns: true or false
Add the arbitrary GAP object *o* as an event to the tracer *t*. This returns true if the event is accepted by the tracer, and false if not. Events are always accepted by recording tracers.

7.1.6 TraceLength (for IsTracer)

- ▷ `TraceLength(t)` (operation)
Returns: An integer
Get the number of events in the tracer *t*.

7.1.7 TraceEvent (for IsTracer, IsPosInt)

- ▷ `TraceEvent(t, i)` (operation)
Returns: A GAP object
Get the event at position *i* in the tracer *t*. The argument *i* must lie in the range `[1..TraceLength(t)]`.

7.1.8 GetEvents (for IsTracer)

- ▷ `GetEvents(t)` (operation)
Returns: A GAP list
Get a list of all events in the tracer.

Chapter 8

Tutorial

8.1 Partition Stacks

A Partition Stack represents an ordered partition which can be modified in two ways:

- A cell in the ordered partition can be split into two pieces
- The ordered partition can be reverted to an earlier state, when it had fewer cells.

Let's look at an example! Firstly, we will create a partition stack which partitions [1..8] (Partition Stacks are always defined on a range of integers starting from 1). In the initial state, the partition has a single cell of size 1.

Example

```
gap> LoadPackage("BacktrackKit", false);  
gap> p := PartitionStack(8);  
[ [ 1, 2, 3, 4, 5, 6, 7, 8 ] ]
```

In most applications of Partition Stacks, we want to record the list of changes which are made. These are stored in an object known as a Tracer. We will make one.

Example

```
gap> t := RecordingTracer();;
```

The main method used for splitting partitions is `PS_SplitCellsByFunction`. This takes a partition stack, a tracer and a function. It splits each cell of the partition by introducing a cell for each different value the function takes.

Example

```
gap> PS_SplitCellsByFunction(p, t, {x} -> x mod 3);  
gap> p;  
[ [ 3, 6 ], [ 2, 5, 8 ], [ 1, 4, 7 ] ]
```

We can further divide the partition with another function

Example

```
gap> PS_SplitCellsByFunction(p, t, {x} -> x mod 2);  
gap> p;  
[ [ 6 ], [ 2, 8 ], [ 4 ], [ 3 ], [ 5 ], [ 1, 7 ] ]
```

From this example we can see that whenever an existing cell is split, the new cell is always placed at the end.

There are a range of functions which can be used to find out information about a partition stack,

Example

```
gap> # Number of cells
> PS_Cells(p);
6
gap> # Cells of size 1, in the order they were created
> PS_FixedCells(p);
[ 1, 4, 5, 3 ]
gap> # Contents of the cells of size 1
> PS_FixedPoints(p);
[ 6, 3, 5, 4 ]
gap> # The contents of a cell, as a slice (a type of list)
> PS_CellSlice(p, 2);
<slice size=2>
gap> AsList(last);
[ 2, 8 ]
```

Note that PS_CellSlice does not make a copy of the list, but gives a "view" inside the partition stack. This means it is fast, but the value will become invalid if another split is made in the Partition Stack. Also, in general the list returned by PS_CellSlice is **NOT** sorted.

We can revert to an earlier state of the partition stack. This includes states we may never have explicitly created, while a cell was being split into pieces.

Example

```
gap> PS_RevertToCellCount(p, 3);
gap> p;
[ [ 3, 6 ], [ 2, 5, 8 ], [ 1, 4, 7 ] ]
gap> PS_RevertToCellCount(p, 2);
gap> p;
[ [ 1, 3, 4, 6, 7 ], [ 2, 5, 8 ] ]
gap> AsList(PS_CellSlice(p, 1));
[ 6, 3, 4, 1, 7 ]
```

8.2 Refiners

In partition backtrack, the search takes a list of refiners, each of which represents a group or coset, and calculates their intersection.

Let us begin by calculating the intersection of two groups, G and H .

Example

```
gap> LoadPackage("BacktrackKit", false);
gap> G := Group((1,2,3),(4,5,6),(1,4)(2,5)(3,6));
gap> H := Group((1,2,3,4,5,6),(2,3)(4,5));
gap> # Need to make a partition stack to search over
> ps := PartitionStack(6);
gap> # We create a refiner for each group
> # We have to give this how many points we want the group to act on
> rg := BTKit_Refiner.InGroup(G);
gap> rh := BTKit_Refiner.InGroup(H);
```

```
gap> # Finally, intersect the groups
> BTKit_SimpleSearch(ps, [rg, rh]);
Group([ (1,2,3)(4,6,5), (1,4)(2,5)(3,6) ])
```

The refiner `BTKit_Refiner.InGroup` represents a group represented as a Permutation Group in GAP. There are many other ways of representing groups. For example, we could calculate a set stabilizer:

Example

```
gap> # This represents the group which stabilizes the set [3,4,5,6]
> ss := BTKit_Refiner.SetStab([3,4,5,6]);
gap> # We can prove this by just "searching" on this group
> BTKit_SimpleSearch(ps, [ss]);
Group([ (5,6), (4,5), (3,4), (1,2) ])
gap> # We can intersect this with G
> BTKit_SimpleSearch(ps, [rg,ss]);
Group([ (4,5,6) ])
gap> # We can also intersect G and H and the set stabilizer at the same time!
> BTKit_SimpleSearch(ps, [rg,rh,ss]);
Group(())
```

Writing a new refiner requires some understanding of the Partition Backtracking framework by Jeffry Leon. We will give a (very brief, and incomplete!) overview here.

In general, a refiner represents either a group or coset. Here we will consider the case where we are building a coset which represents mapping some permutation P to another permutation Q under conjugation. This is a group when $P = Q$.

To begin, we will just consider the case where $P = Q$. Then, we need to provide a GAP function R which, given an ordered partition ps on $\{1..n\}$ returns a function $f : \{1..n\} \rightarrow O$ (where O is any valid GAP object) which satisfies the following conditions:

- Invariance: Given a permutation g such that $P^g = g$, then $R(ps^g) = R(ps)^g$. Here, we define ps^g as applying g to each point in each cell of ps , and we define $(R(ps)^g)(x) := R(ps)(x^g)$.

(TO COMPLETE)

References

- [Leo91] Jeffrey S. Leon. Permutation group algorithms based on partitions. I. Theory and algorithms. *J. Symbolic Comput.*, 12(4-5):533–583, 1991. [3](#)
- [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. [3](#)

Index

AddEvent
 for IsTracer, IsObject, [22](#)
ApplyFilters
 for IsBTKitState, IsTracer, IsObject, [6](#)

Backtrack, [7](#)
BacktrackableStateFamily, [5](#)
BTKitStateType, [5](#)
BTKit_BuildProblem, [7](#)
BTKit_ResetStats, [4](#)
BTKit_SimpleAllPermSearch, [4](#)
BTKit_SimpleSearch, [4](#)
BTKit_SimpleSinglePermSearch, [4](#)
BuildRBase, [7](#)

CanonicalisingTracerFromTracer, [22](#)
ConsolidateState
 for IsBacktrackableState, IsTracer, [6](#)
Constraint, [9](#)
Constraint.Centralise, [12](#)
Constraint.Centralize, [12](#)
Constraint.Conjugate, [13](#)
Constraint.Everything, [14](#)
Constraint.InCoset, [10](#)
Constraint.InGroup, [10](#)
Constraint.InLeftCoset, [11](#)
Constraint.InRightCoset, [10](#)
Constraint.IsEven, [13](#)
Constraint.IsOdd, [14](#)
Constraint.IsTrivial, [14](#)
Constraint.LargestMovedPoint, [13](#)
Constraint.MovedPoints, [13](#)
Constraint.None, [14](#)
Constraint.Normalise, [12](#)
Constraint.Normalize, [12](#)
Constraint.Stabilise, [11](#)
Constraint.Stabilize, [11](#)
Constraint.Transport, [11](#)

DummyRefiner
 for IsConstraint, [17](#)

FinaliseRBaseForConstraints, [6](#)
FirstFixedPoint, [7](#)
FollowingTracer, [22](#)

GetEvents
 for IsTracer, [23](#)

InfoBTKit, [4](#)
InitialiseConstraints, [6](#)
IsBacktrackableState
 for IsObject, [5](#)
IsBTKitState
 for IsBacktrackableState, [5](#)
IsPartitionStack
 for IsObject, [18](#)
IsTracer
 for IsObject, [22](#)

PartitionStack, [18](#)
PS_AsPartition
 for IsPartitionStack, [19](#)
PS_CellLen
 for IsPartitionStack, IsPosInt, [19](#)
PS_CellOfPoint
 for IsPartitionStack, IsPosInt, [20](#)
PS_Cells
 for IsPartitionStack, [19](#)
PS_CellSlice
 for IsPartitionStack, IsPosInt, [19](#)
PS_Extend
 for IsPartitionStack, IsPosInt, [18](#)
PS_ExtendedPoints
 for IsPartitionStack, [18](#)
PS_ExtendedSplitCellsByFunction
 for IsPartitionStack, IsTracer, IsFunction, [20](#)
PS_Fixed
 for IsPartitionStack, [19](#)
PS_FixedCells

- for IsPartitionStack, [19](#)
- PS_FixedPoints
 - for IsPartitionStack, [19](#)
- PS_Points
 - for IsPartitionStack, [18](#)
- PS_RevertToCellCount
 - for IsPartitionStack, IsPosInt, [20](#)
- PS_SplitCellByFunction
 - for IsPartitionStack, IsTracer, IsPosInt, Is-
Function, [20](#)
- PS_SplitCellByUnorderedFunction
 - for IsPartitionStack, IsTracer, IsPosInt, Is-
Function, [20](#)
- PS_SplitCellsByFunction
 - for IsPartitionStack, IsTracer, IsFunction, [20](#)
- PS_SplitCellsByUnorderedFunction
 - for IsPartitionStack, IsTracer, IsFunction, [21](#)
- RecordingTracer, [22](#)
- RefineConstraints, [6](#)
- RestoreState
 - for IsBacktrackableState, IsObject, [5](#)
- SaveState
 - for IsBacktrackableState, [5](#)
- TraceEvent
 - for IsTracer, IsPosInt, [23](#)
- TraceLength
 - for IsTracer, [23](#)