WPE

Provides efficient methods for working with wreath product elements.

0.8

21 October 2024

Friedrich Rober

Friedrich Rober
Email: friedrich.rober@rwth-aachen.de

Contents

1 Introduction **3**
- 1.1 Overview . . . 3
- 1.2 Intuitive Research . . . 4
- 1.3 Efficient Computing . . . 5

2 Notation **7**
- 2.1 Wreath Products . . . 7
- 2.2 Wreath Cycles . . . 7
- 2.3 Sparse Wreath Cycles . . . 8

3 Tutorial **10**
- 3.1 Creating Wreath Product Elements . . . 10
- 3.2 Displaying Wreath Product Elements . . . 11
- 3.3 Computing in Wreath Products . . . 13
- 3.4 Conjugacy Problem . . . 14
- 3.5 Conjugacy Classes . . . 14
- 3.6 Centralizer . . . 15
- 3.7 Cycle Index Polynomial . . . 16

4 Functions **18**
- 4.1 Generic Wreath Product Representation . . . 18
- 4.2 Accessing Components . . . 19
- 4.3 Properties of Wreath Product Elements . . . 19
- 4.4 Printing, Viewing and Displaying . . . 20
- 4.5 Cycle Index of Wreath Products . . . 21

5 Operations **23**
- 5.1 Operations List . . . 23

References **25**

Index **26**

Chapter 1

Introduction

This chapter serves as an introduction and showcases some highlights of the package WPE.

1.1 Overview

The package WPE (*W*reath *P*roduct *E*lements). provides methods to work with elements of finite groups which are wreath products. It contributes to *intuitive research* and *efficient computing* in wreath products of groups.

It allows access to a representation of wreath products, which we refer to as the *generic representation*, that is more intuitive to the User when working with wreath products of groups. Access is as straight-forward as using the provided command `IsomorphismWreathProduct` on a wreath product created by the native GAP command `WreathProduct`, see 1.2 for an example.

Additionally, this representation may have computational benefits over other representations. Note, that just by loading the package WPE and without any additional setup, all optimizations are applied to computations in wreath products created by the native GAP command `WreathProduct`, by exploiting the generic representation under the hood when appropiate. See 1.3 for a highlight showcase of such computational problems.

In particular, this package provides efficient methods for finding *conjugating elements*, *conjugacy classes*, and *centralisers* in wreath products. The implementations are based on an accompanying publication [BNRW22], that generalizes results from [Spe32] and [Ore42] on monomial groups, wreath products whose top group is the full symmetric group.

For example, the computation of all 886640 conjugacy classses of elements of the wreath product $W = \mathrm{M}_{22} \wr \mathrm{A}_9$ takes about 12 seconds with WPE. With native GAP this computation is not feasible.

Example

```
gap> LoadPackage("WPE");;
gap> K := MathieuGroup(22);;
gap> H := AlternatingGroup(9);;
gap> G := WreathProduct(K, H);;
gap> C := ConjugacyClasses(G);;
gap> Size(C);
886640
```

1.2 Intuitive Research

One of the two main goals of the package is to provide the User with tools to conduct *intuitive research* in wreath products of groups on the computer.

In this section we present an example session which demonstrates how we can access the generic representation of a wreath product. As noted in the introduction, no additional setup is required if one wants to benefit from the optimizations for computations in wreath products (see 1.3 for examples on this).

First we construct the wreath product $G = \mathrm{Alt}(5) \wr \mathrm{Sym}(7)$ (see 2.1). For this we use the native GAP command `WreathProduct` (**Reference: WreathProduct**). The resulting group is embedded into a symmetric group on $5 \cdot 7 = 35$ points via the imprimitive action of the wreath product. The size of the group is

$$|G| = |\mathrm{Alt}(5)|^7 \cdot |\mathrm{Sym}(7)| = 60^7 \cdot 5040 = 14\,108\,774\,400\,000\,000\,.$$

Example

```
gap> K := AlternatingGroup(5);;
gap> H := SymmetricGroup(7);;
gap> G := WreathProduct(K, H);
<permutation group of size 14108774400000000 with 4 generators>
```

Now we construct an isomorphism to a wreath product given in generic representation that is provided in WPE. For this, we need to load the package WPE.

Example

```
gap> LoadPackage("WPE");;
gap> iso := IsomorphismWreathProduct(G);;
gap> W := Image(iso);
<group of size 14108774400000000 with 4 generators>
```

Let us compare how GAP displays elements of G and W respectively. Elements of G are represented as permutations. In this representation it is hard to identify the base and top components of this element (see 2.1).

Example

```
gap> g := (1,13,3,14,4,12,2,15,5,11)
>         (6,31,21,7,35,25,9,33,23,8,34,24,10,32,22)
>         (18,19,20);;
gap> g in G;
true
```

Elements of W however are represented as generic wreath product elements (see 2.1). This allows us to read off the base and top component of the element easily by either printing or displaying the element. Otherwise, by default the element is viewed in compressed form (see 4.4). This printing behaviour is similar to the behaviour of matrices in GAP.

Example

```
gap> w := g ^ iso;
< wreath product element with 7 base components >
gap> Print(w);
[ (1,3,4,2,5), (2,5)(3,4), (), (3,4,5), (1,2)(4,5), (), (), (1,3)(2,7,5) ]
gap> Display(w);
       1          2       3      4         5       6   7       top
( (1,3,4,2,5), (2,5)(3,4), (), (3,4,5), (1,2)(4,5), (), (); (1,3)(2,7,5) )
```

Furthermore, we can display and access each component easily with the provided commands.

Example

```
gap> BaseComponentOfWreathProductElement(w, 2);
(2,5)(3,4)
gap> TopComponentOfWreathProductElement(w);
(1,3)(2,7,5)
```

1.2.1 The Power of Component-wise Representation

This component-wise representation is often exactly the one that we encounter in research on wreath products. Thus having it available on the computer greatly sharpens our intuition. We can make very non-trivial statements by looking at the components of such an element, and for the case of the element w even without a computer.

Let us start off with an easy observation. Just by looking at the top component of w, i.e. $(1,3)(2,7,5)$, we can see that the smallest power of w that lies in the base group of W has exponent 6, since it has to be equal to the order of the top component.

Example

```
gap> m := Order(w);
30
gap> First( [1 .. m], k -> IsOne(TopComponentOfWreathProductElement(w ^ k)) );
6
gap> Display(w ^ 6);
       1            2            3       4       5       6       7       top
( (1,2,3,5,4), (1,4,5,2,3), (1,2,3,5,4), (), (1,3,2,5,4), (), (1,3,2,5,4); () )
```

Now let us be more advanced. Just by looking at the element w, we can deduce structural information on the conjugacy class w^W. All elements conjugate to w in $W = K \wr H$ must have at least one trivial base component, since the territory of w (see 2.2) contains exactly six elements, whereas the top group acts on seven points.

Example

```
gap> Length(Territory(w));
6
gap> NrMovedPoints(H);
7
```

On the other hand, all such elements must have at least three non-trivial base components, since the wreath cycle decomposition of w (see 2.2) contains exactly three wreath cycles with non-trivial yades (see 2.3).

Example

```
gap> Number(WreathCycleDecomposition(w), c -> not IsOne(Yade(c)));
3
```

Moreover, for each integer k with $3 \leq k \leq 6$ there exists at least one conjugate element with exactly k non-trivial base components.

1.3 Efficient Computing

One of the two main goals of the package is to empower the User to carry out *efficient computations* in wreath products of groups on the computer.

In this section we present a highlight showcase of computational problems that benefit from the generic representation. As noted in the introduction, no additional setup is required if one wants to benefit from the optimizations for computations in wreath products. We simply create the wreath products via the native GAP command `WreathProduct` (**Reference: WreathProduct**), and the generic representation provided by WPE is used under the hood whenever appropiate.

We only give a summary of some computational problems that now become approachable on the computer, and include examples for such computations in Chapter 3 containing extensive GAP sessions that can be followed like a tutorial.

In the following let $G = K \wr H$ be a wreath product of finite groups, where $H \leq \mathrm{Sym}(m)$. Further let $x, y \in P = K \wr \mathrm{Sym}(m)$ be elements of the parent wreath product P which is given in the same representation as G.

Conjugacy Problem

Solve the conjugacy problem for x and y over G, i.e. decide whether there exists $c \in G$ with $x^c = y$ and if it does, explicitly compute such a conjugating element c.

Conjugacy Classes

Enumerate representatives of all conjugacy classes of elements of G, i.e. return elements $g_1, \ldots, g_\ell$ such that $g_1^G, \ldots, g_\ell^G$ are the conjugacy classes of G.

Centralizer

Compute the centralizer of x in G, i.e. compute a generating set of $C_G(x)$.

Cycle Index Polynomial

Compute the cycle index polynomial of G either for the imprimitive action or the product action.

Chapter 2

Notation

This chapter explains the notation of the package WPE, mainly influenced by the accompanying publication [BNRW22].

2.1 Wreath Products

Let $G = K \wr H$ be a wreath product of two groups, where H is a permutation group of degree m. The wreath product is defined as the semidirect product of the function space K^m with H, where $\pi \in H$ acts on $f \in K^m$ by setting $f^\pi : \{1,\dots,m\} \to K, i \mapsto [(i)\pi^{-1}]f$. Note that G naturally embeds into the *parent wreath product*, that is $P = K \wr \mathrm{Sym}(m) \geq G$.

Formally we can write an element of G as a tuple $g = (f,\pi) \in G$, where $f \in K^m$ is a function $f : \{1,\dots,m\} \to K$ and $\pi \in H \leq \mathrm{Sym}(m)$ is a permutation on m points. We call f the *base component* and π the *top component* of g.

We can naturally identify a map $f \in K^m$ with a tuple $(g_1,\dots,g_m)$, where each $g_i \in K$ is the image of $i \in \{1,\dots,m\}$ under f. This yields a second useful notation for elements in G by writing $g = (g_1,\dots,g_m;\pi)$. Note that we use a semicolon to seperate the base component from the top component. Further we call the element g_i the *i-th base component* of g.

Analogously, the subgroup $B = K^m \times \langle 1_H \rangle \leq G$ is called the *base group* of G and the subgroup $T = \langle 1_K \rangle^m \times H \leq G$ is called the *top group* of G.

With the above notation, the multiplication of two elements

$$g = (f,\pi) = (g_1,\dots,g_m;\pi), h = (d,\sigma) = (h_1,\dots,h_m;\sigma)$$

of $G = K \wr H$, a wreath product of finite groups, can be written as

$$g \cdot h = (f \cdot d^{(\pi^{-1})}, \pi \cdot \sigma) = (g_1 \cdot h_{1\pi},\dots,g_m \cdot h_{m^\pi};\pi \cdot \sigma).$$

2.2 Wreath Cycles

In a permutation group we have the well-known concept of a cycle decomposition. For wreath products we have a similar concept called *wreath cycle decomposition* that allows us to solve certain computational tasks more efficiently.

Detailed information on *wreath cycle decompositions* can be found in Chapter 2 in [BNRW22]. Chapters 3-5 in [BNRW22] describe how these can be exploited for finding conjugating elements,

conjugacy classes, and centralisers in wreath products, and Chapter 6 in [BNRW22] contains a table of timings of sample computations done with WPE vs. native GAP.

We use the notation from Section 2.1 in order to introduce the following concepts.

Definition : We define the *territory* of an element $g = (g_1, \ldots, g_m; \pi) \in G$ by $\mathrm{terr}(g) := \mathrm{supp}(\pi) \cup \{i : g_i \neq 1\}$, where $\mathrm{supp}(\pi)$ denotes the set of moved points of π.

Definition : Two elements $g, h \in G$ are said to be *disjoint* if their territories are disjoint.

Lemma : Disjoint elements in G commute.

Definition : An element $g = (g_1, \ldots, g_m; \pi) \in G$ is called a *wreath cycle* if either π is a cycle in $\mathrm{Sym}(n)$ and $\mathrm{terr}(g) = \mathrm{supp}(\pi)$, or $|\mathrm{terr}(g)| = 1$.

Example : For example, if we consider the wreath product $\mathrm{Sym}(4) \wr \mathrm{Sym}(5)$, the element

$$(\ (), (1,2,3), (), (1,2), (); (1,2,4)\ )$$

is a wreath cycle as described in the first case and the element

$$(\ (), (), (1,3), (), (); ()\ )$$

is a wreath cycle as described in the second case. Moreover, these elements are disjoint and thus commute.

Theorem : Every element of G can be written as a finite product of disjoint wreath cycles in P. This decomposition is unique up to ordering of the factors. We call such a decomposition a *wreath cycle decomposition*.

2.3 Sparse Wreath Cycles

We use the notation from Section 2.1 in order to introduce the following concepts.

The main motivation for introducing the concept of *sparse wreath cycles* is the efficient computation of centralisers of wreath product elements. Simply put, we compute the centraliser $C_G(g)$ of an arbitrary element $g \in P$ in G by conjugating it in P to a restricted representative $h = g^c \in P$, computing the centraliser of h in G and then conjugating it back. The wreath cycle decomposition of the representative h consists only of sparse wreath cycles.

More information on *sparse wreath cycles* and centralisers of wreath product elements can be found in Chapter 5 in [BNRW22].

Definition : We say that a wreath cycle $g = (g_1, \ldots, g_m; \pi) \in G$ is a *sparse wreath cycle*, if there exists an i_0 such that $g_i = 1$ for all $i \neq i_0$.

Example : For example, if we consider the wreath product $\mathrm{Sym}(4) \wr \mathrm{Sym}(5)$, the element

$$(\ (), (1,2,3), (), (), (); (1,2,4)\ )$$

is a sparse wreath cycle, as well as the element

$$(\ (), (), (1,3), (), (); ()\ ).$$

A very important invariant under conjugation is the *yade* of a wreath cycle.

Definition : For a wreath cycle $g = (f, \pi) \in G$ and a point $i \in \mathrm{terr}(g)$ we define the *yade* of g in i as

$$[(i)\pi^0]f \cdot [(i)\pi^1]f \cdots [(i)\pi^{|\pi|-1}]f\ .$$

Example : Consider the wreath product $\mathrm{Sym}(4) \wr \mathrm{Sym}(5)$, and the wreath cycle

$$g = (f, \pi) = (\ (), (1,2,3), (), (1,2), (); (1,2,4)\ ).$$

The yade evaluated at $i = 1$ is given by

$$[(1)\pi^0]f \cdot [(1)\pi^1]f \cdot [(1)\pi^2]f = [1]f \cdot [2]f \cdot [4]f = () \cdot (1,2,3) \cdot (1,2) = (2,3)$$

and the yade evaluated at $j = 4$ is given by

$$[(4)\pi^0]f \cdot [(4)\pi^1]f \cdot [(4)\pi^2]f = [4]f \cdot [1]f \cdot [2]f = (1,2) \cdot () \cdot (1,2,3) = (1,3)\ .$$

Up to conjugacy, the yade is independent under the chosen evaluation point i. Moreover, wreath cycles are conjugate over G if and only if the top components are conjugate over H and the yades are conjugate over K. More specific, we can conjugate a wreath cycle g to a sparse wreath cycle h such that the i-th base component of h contains the yade of g in i. This leads to the following result.

Theorem : Every element $g \in P$ can be conjugated by some $c \in K^m \times \langle 1_H \rangle \leq P$ to an element $h = g^c \in P$ such that the wreath cycle decomposition of h consists only of sparse wreath cycles.

Chapter 3

Tutorial

This chapter is a collection of tutorials that show how to work with wreath products in GAP in conjunction with the package WPE.

3.1 Creating Wreath Product Elements

In this section we present an example session which demonstrates how we can create wreath products elements by specifying its components.

In the following we will work with the wreath product $G = \mathrm{Alt}(5) \wr \mathrm{Sym}(4)$.

Example

```
gap> LoadPackage("WPE");;
gap> K := AlternatingGroup(5);;
gap> H := SymmetricGroup(4);;
gap> G := WreathProduct(K, H);
<permutation group of size 311040000 with 10 generators>
```

The resulting group G is embedded into a symmetric group on $5 \cdot 4 = 20$ points via the imprimitive action of the wreath product. The size of the group is

$$|G| = |\mathrm{Alt}(5)|^4 \cdot |\mathrm{Sym}(4)| = 60^4 \cdot 24 = 311\,040\,000\,.$$

Suppose we would like to input the wreath product element

$$g = (\ (1,5,2,4,3), (1,3,5,2,4), (1,5,3,4,2), (1,4,5);\ (1,3)(2,4)\ )$$

as an element of G. The method `WreathProductElementList` is the preferred way to create a wreath product element by specifying its components. Note that we first specify the four base components and at the end the top component as the last entry.

Example

```
gap> gList := [ (1,5,2,4,3), (1,3,5,2,4), (1,5,3,4,2), (1,4,5), (1,3)(2,4) ];;
gap> g := WreathProductElementList(G, gList);
(1,15,3,11,5,12)(2,14)(4,13)(6,18,8,20)(7,19,10,17)(9,16)
gap> g in G;
true
```

On the other hand, the method ListWreathProductElement can be used to get a list containing the components of a wreath product element.

Example

```
gap> ListWreathProductElement(G, g);
[ (1,5,2,4,3), (1,3,5,2,4), (1,5,3,4,2), (1,4,5), (1,3)(2,4) ]
gap> last = gList;
true
```

The package author has implemented the methods ListWreathProductElement (**Reference: ListWreathProductElement**) and WreathProductElementList (**Reference: WreathProductElementList**) in GAP in order to translate between list representations of wreath product elements and other representations. The naming conventions are the same as for ListPerm and PermList.

Moreover, all functions that work for IsWreathProductElement can also be used on these list representations. However, it is not checked if the list indeed represents a wreath product element.

Example

```
gap> Territory(gList);
[ 1, 2, 3, 4 ]
```

If the wreath product element is "sparse", i.e. has only a few non-trivial components, it might be easier to create it by embedding its non-trivial components into G directly and multiplying them. Note however, that WreathProductElementList might be faster as it avoids group multiplications.

Example

```
gap> h := (1,2,3) ^ Embedding(G,2)
>       * (1,5,2,4,3) ^ Embedding(G,4)
>       * (1,2,4) ^ Embedding(G, 5);
(1,6,17,4,9,19,3,8,16,5,10,20,2,7,18)
gap> hList := ListWreathProductElement(G, h);
[ (), (1,2,3), (), (1,5,2,4,3), (1,2,4) ]
gap> IsWreathCycle(hList);
true
```

3.2 Displaying Wreath Product Elements

In this section we present an example session which demonstrates how we can display wreath product elements in an intuitive way. Wreath product elements are viewed, printed and displayed (see section (**Reference: View and Print**) for the distinctions between these operations) as generic wreath product elements (see section 2.1).

Suppose we are given some element g in the wreath product $G = \mathrm{Alt}(5) \wr \mathrm{Sym}(4)$, and would like to view its components in a nice way.

Example

```
gap> LoadPackage("WPE");;
gap> K := AlternatingGroup(5);;
gap> H := SymmetricGroup(4);;
gap> G := WreathProduct(K, H);;
gap> iso := IsomorphismWreathProduct(G);;
gap> W := Image(iso);;
gap> g := (1,15,8,20)(2,14,7,19,5,12,6,18,3,11,10,17)(4,13,9,16);;
gap> g in G;
true
```

First we translate the element g into a generic wreath product element w. GAP uses `ViewObj` to print w in a compressed form.

Example

```
gap> w := g ^ iso;
< wreath product element with 4 base components >
```

If we want to print this element in a "machine-readable" way, we could use one of the following methods.

Example

```
gap> Print(w);
[ (1,5,2,4,3), (1,3,5,2,4), (1,5,3,4,2), (1,4,5), (1,3,2,4) ]
gap> L := ListWreathProductElement(W, w);
[ (1,5,2,4,3), (1,3,5,2,4), (1,5,3,4,2), (1,4,5), (1,3,2,4) ]
gap> L = ListWreathProductElement(G, g);
true
```

Usually, we want to display this element in a nice format instead, which is "human-readable" and allows us to quickly distinguish components.

Example

```
gap> Display(w);
       1           2           3          4        top
( (1,5,2,4,3), (1,3,5,2,4), (1,5,3,4,2), (1,4,5); (1,3,2,4) )
```

There are many display options available for adjusting the display behaviour for wreath product elements to your liking (see 4.4). For example, we might want to display the element vertically. We can do this for a single call to the ‘Display‘ command without changing the global display options like this:

Example

```
gap> Display(w, rec(horizontal := false));
  1: (1,5,2,4,3)
  2: (1,3,5,2,4)
  3: (1,5,3,4,2)
  4: (1,4,5)
top: (1,3,2,4)
gap> Display(w);
       1           2           3          4        top
( (1,5,2,4,3), (1,3,5,2,4), (1,5,3,4,2), (1,4,5); (1,3,2,4) )
```

We can also change the global display options via the following command.

Example

```
gap> SetDisplayOptionsForWreathProductElements(rec(horizontal := false));
gap> Display(w);
  1: (1,5,2,4,3)
  2: (1,3,5,2,4)
  3: (1,5,3,4,2)
  4: (1,4,5)
top: (1,3,2,4)
```

All changes to the global behaviour can be reverted to the default behaviour via the following command.

Example
```
gap> ResetDisplayOptionsForWreathProductElements();
gap> Display(w);
       1           2           3          4        top
( (1,5,2,4,3), (1,3,5,2,4), (1,5,3,4,2), (1,4,5); (1,3,2,4) )
```

But sometimes, it is sufficient to just look at some components of a wreath product element. We can directly use the list representation to access the components on a low-level or we can use high-level functions on wreath product elements instead.

Example
```
gap> a := BaseComponentOfWreathProductElement(w, 3);
(1,5,3,4,2)
gap> a = L[3];
true
gap> b := TopComponentOfWreathProductElement(w);
(1,3,2,4)
gap> b = L[5];
true
```

3.3 Computing in Wreath Products

As noted in the introduction, no additional setup is required if one wants to benefit from the optimizations for computations in wreath products. We simply create the wreath products via the native GAP command `WreathProduct` (**Reference: WreathProduct**), and the generic representation is used under the hood whenever appropiate.

We include in the following sections examples for each computational problem listed in 1.3. For all such examples we fix the following wreath product.

Example
```
gap> LoadPackage("WPE");;
gap> K := Group([ (1,2,3,4,5), (1,2,4,3) ]);; # F(5)
gap> H := Group([ (1,2,3,4,5,6), (2,6)(3,5) ]);; # D(12)
gap> G := WreathProduct(K, H);
<permutation group of size 768000000 with 4 generators>
gap> P := WreathProduct(K, SymmetricGroup(NrMovedPoints(H)));
<permutation group of size 46080000000 with 4 generators>
gap> IsSubgroup(P, G);
true
gap> iso := IsomorphismWreathProduct(P);;
```

Moreover, we fix the following elements of the parent wreath product P. We choose them in such a way, that they do not lie in the smaller wreath product G for demonstration purposes only.

Example
```
gap> x := (1,23,12,6,4,24,13,9,5,21,15,10,2,25,14,7)(3,22,11,8)(16,30,20,28)(17,27,19,26)(18,29);
gap> y := (1,12,26,8,3,14,28,7,2,13,27,10,5,11,30,6)(4,15,29,9)(16,23,20,22)(17,24,19,21)(18,25);
gap> Display(x ^ iso);
      1          2          3          4          5          6          top
( (1,3,2,5), (1,4,5,2), (1,3,4,2), (1,5,3,4), (1,5,4,3,2), (1,2,4,3); (1,5,3,2)(4,6) )
gap> Display(y ^ iso);
       1       2        3           4          5          6          top
( (1,2,3,4,5), (), (1,5,4,3,2), (1,3,5,2,4), (1,2)(3,5), (1,3,2,5); (1,3,6,2)(4,5) )
```

```
gap> x in P and y in P;
true
gap> not x in G and not y in G;
true
```

3.4 Conjugacy Problem

We now demonstrate how to solve the conjugacy problem for x and y over G, i.e. decide whether there exists $c \in G$ with $x^c = y$ and if it does, explicitly compute such a conjugating element c.

We continue the GAP session from Section 3.3.

To check in GAP whether two elements are conjugate in a group we use native GAP command `RepresentativeAction` (**Reference: RepresentativeAction**).

Example

```
gap> RepresentativeAction(G, x, y);
fail
```

The output `fail` indicates, that x and y are not conjugate over G. But are x and y conjugate in the parent wreath product?

Example

```
gap> c := RepresentativeAction(P, x, y);
(2,5)(3,4)(6,8,9,7)(11,29,25)(12,26,21,13,28,22,15,27,24,14,30,23)
gap> Display(c^iso);
      1           2           3      4       5           6        top
( (2,5)(3,4), (1,3,4,2), (1,4,5,2), (), (1,3,2,5), (2,4,5,3); (3,6,5) )
gap> x ^ c = y;
true
```

We see, that indeed these elements are conjugate over the larger wreath product P by the conjugating element $c \in P$.

3.5 Conjugacy Classes

Enumerate representatives of all conjugacy classes of elements of G, i.e. return elements $g_1, \ldots, g_\ell$ such that $g_1^G, \ldots, g_\ell^G$ are the conjugacy classes of G.

We continue the GAP session from Section 3.3. In particular recall the definition of the isomorphism `iso`.

To compute in GAP the conjugacy classes of elements of a group we use `ConjugacyClasses` (**Reference: ConjugacyClasses attribute**).

Example

```
gap> CC := ConjugacyClasses(G);;
gap> Length(CC);
1960
```

We see that there are 1 960 many conjugacy classes of elements of G. Let us look at a single conjugacy class.

Example

```
gap> A := CC[1617];
(2,4,5,3)(6,26)(7,29,9,30,10,28,8,27)(11,21)(12,22)(13,23)(14,24)(15,25)^G
```

We can compute additional information about a conjugacy class on the go. For example, we can ask GAP for the number of elements in this class.

Example

```
gap> Size(A);
60000
```

To access the representative of this class, we do the following.

Example

```
gap> a := Representative(A);
(2,4,5,3)(6,26)(7,29,9,30,10,28,8,27)(11,21)(12,22)(13,23)(14,24)(15,25)
gap> Display(a ^ iso);
      1          2      3   4   5   6      top
( (2,4,5,3), (2,4,5,3), (), (), (), (); (2,6)(3,5) )
```

Representatives are always given in a sparse format, e.g. all cycles in the wreath cycle decomposition of a are sparse (see 2.3).

3.6 Centralizer

Compute the centralizer of x in G, i.e. compute a generating set of $C_G(x)$.

We continue the GAP session from 3.3. In particular recall the definition of the isomorphism `iso`.

To compute in GAP the centralizer of an element in a group we use `Centralizer` (**Reference: centraliser**).

Example

```
gap> C := Centralizer(G, x);
Group([ (16,20)(17,19)(26,27)(28,30), (16,19,20,17)(26,28,27,30),
        (1,4,5,2)(6,9,10,7)(12,13,15,14)(21,25,23,24) ])
```

We can compute additional information about the centralizer on the go. For example, we can ask GAP for the number of elements in G that centralize x.

Example

```
gap> Size(C);
16
```

The generators of a centralizer are always given in a sparse format, e.g. all cycles in the wreath cycle decomposition of a generator g are sparse (see 2.3).

Example

```
gap> for g in GeneratorsOfGroup(C) do
>   Display(g ^ iso);
> od;
  1   2   3       4       5       6       top
( (), (), (), (1,5)(2,4), (), (1,2)(3,5); () )

  1   2   3       4       5       6      top
( (), (), (), (1,4,5,2), (), (1,3,2,5); () )

      1          2          3      4       5      6   top
( (1,4,5,2), (1,4,5,2), (2,3,5,4), (), (1,5,3,4), (); () )
```

3.7 Cycle Index Polynomial

Compute the cycle index polynomial of G either for the imprimitive action or the product action. We do not continue the GAP session from 3.3 since the wreath product is too large to make sense of the cycle index polynomial just by looking at it. Instead we use the following wreath product.

Example

```
gap> LoadPackage("WPE");;
gap> K := Group([ (1,2), (1,2,3) ]);; # S(3)
gap> H := Group([ (1,2) ]);; # C(2)
gap> G_impr := WreathProduct(K, H);;
gap> NrMovedPoints(G_impr);
6
gap> Order(G_impr);
72
```

To compute in GAP the cycle index of a a group we use `CycleIndex` (**Reference: CycleIndex**). Note that by default, the wreath product is given in imprimitive action.

Example

```
gap> c_impr := CycleIndex(G_impr);
1/72*x_1^6+1/12*x_1^4*x_2+1/18*x_1^3*x_3+1/8*x_1^2*x_2^2+1/6*x_1*x_2*x_3
+1/12*x_2^3+1/4*x_2*x_4+1/18*x_3^2+1/6*x_6
```

For example, the second monomial `1/12*x_1^4*x_2` tells us that there are exactly $\frac{72}{12} = 6$ elements with cycle type $(4,1)$, i.e. elements that have four fixed points and one 2-cycle. If one wants to access these monomials on the computer, one needs to use `ExtRepPolynomialRatFun` (**Reference: ExtRepPolynomialRatFun**).

Example

```
gap> Display(ExtRepPolynomialRatFun(c_impr));
[ [ 6, 1 ], 1/6, [ 3, 2 ], 1/18, [ 2, 1, 4, 1 ], 1/4, [ 2, 3 ], 1/12,
  [ 1, 1, 2, 1, 3, 1 ], 1/6, [ 1, 2, 2, 2 ], 1/8, [ 1, 3, 3, 1 ], 1/18,
  [ 1, 4, 2, 1 ], 1/12, [ 1, 6 ], 1/72 ]
```

The way how to read this representation is roughly the following. The list consists of alternating entries, the first one encoding the monomial and the second one the corresponding coefficient, for example consider `[ 1, 4, 2, 1 ], 1/12`. The coefficient is `1/12` and the monomial is encoded by `[ 1, 4, 2, 1 ]`. The encoding of the monomial again consists of alternating entries, the first one encoding the indeterminant and the second one its exponent. For example `[ 1, 4, 2, 1 ]` translates to `x_1^4 * x_2^1`. For more details, see (**Reference: The Defining Attributes of Rational Functions**).

If we want to consider the wreath product given in product action, we need to use the command `WreathProductProductAction` (**Reference: WreathProductProductAction**)

Example

```
gap> G_prod := WreathProductProductAction(K, H);;
gap> NrMovedPoints(G_prod);
9
gap> c_prod := CycleIndex(G_prod);
1/72*x_1^9+1/6*x_1^3*x_2^3+1/8*x_1*x_2^4+1/4*x_1*x_4^2+1/9*x_3^3+1/3*x_3*x_6
```

However, we do not need to create the wreath product in order to compute the cycle index of the group. Thus the package provides the functions `CycleIndexWreathProductImprimitiveAction`

(4.5.1) and `CycleIndexWreathProductProductAction` (4.5.2) to compute the cycle index directly from the components K and H without writing down a representation of $K \wr H$.

Example

```
gap> c1 := CycleIndexWreathProductImprimitiveAction(K, H);;
gap> c_impr = c1;
true
gap> c2 := CycleIndexWreathProductProductAction(K, H);;
gap> c_prod = c2;
true
```

Chapter 4

Functions

Here we include a list of all functions that are provided to the User.

The following functions are designed to improve the User experience when working or experimenting with wreath products of finite groups and their elements. Most functions are about presenting elements in an intuitive way and giving access to useful information.

4.1 Generic Wreath Product Representation

The main way for the User to look at a wreath product in a "human-readable" way is by using an isomorphism from a specialised wreath product representation to a generic representation.

4.1.1 IsomorphismWreathProduct

▷ `IsomorphismWreathProduct(G)` (operation)

returns an isomorphism from a specialized wreath product `G` to a generic wreath product.

Example

```
gap> K := AlternatingGroup(5);;
gap> H := SymmetricGroup(4);;
gap> G := WreathProduct(K, H);
<permutation group of size 311040000 with 10 generators>
gap> iso := IsomorphismWreathProduct(G);;
gap> W := Image(iso);
<group of size 311040000 with 4 generators>
```

For an overview on wreath product representations in GAP see 5.1.1.

In the background, it uses the low-level functions `ListWreathProductElement` and `WreathProductElementList` and wraps the `IsList` representations into `IsWreathProductElement` representations.

For performant code, we recommend to use these low-level functions instead of `IsomorphismWreathProduct`. All functions for `IsWreathProductElement` also work on `IsList` objects that represent a wreath product element. However, it is not checked that the `IsList` object actually represents a wreath product element.

4.2 Accessing Components

The following functions give access to components of wreath products and their elements.

4.2.1 ComponentsOfWreathProduct

▷ `ComponentsOfWreathProduct(W)` (function)

returns a list of two groups [K, H], where `W = K wr H`. The argument `W` must be a wreath product (see 2.1).

4.2.2 TopGroupOfWreathProduct

▷ `TopGroupOfWreathProduct(W)` (function)

returns a group, namely the top group $\langle 1_K \rangle^m \times H$ of the wreath product $W = K \wr H$ (see 2.1).

4.2.3 BaseGroupOfWreathProduct

▷ `BaseGroupOfWreathProduct(W[, i])` (function)

returns a group, namely the base group $K^m \times \langle 1_H$ of the wreath product $W = K \wr H$. If the optional argument `i` is provided, the function returns the `i`-th factor of the base group of `W` (see 2.1).

4.2.4 TopComponentOfWreathProductElement

▷ `TopComponentOfWreathProductElement(x)` (function)

returns a group element, namely the top component of `x`. The argument `x` must be a wreath product element (see 2.1).

4.2.5 BaseComponentOfWreathProductElement

▷ `BaseComponentOfWreathProductElement(x[, i])` (function)

returns a group element, namely the base component of `x`. If the optional argument `i` is provided, the function returns the `i`-th base component of `x`. The argument `x` must be a wreath product element and the optional argument `i` must be an integer (see 2.1).

4.3 Properties of Wreath Product Elements

The following functions give access to important properties of wreath product elements.

4.3.1 Territory

▷ `Territory(x)` (attribute)

returns a list, namely the territory of `x`. The argument `x` must be a wreath product element (see 2.2).

4.3.2 IsWreathCycle

▷ `IsWreathCycle(x)` (attribute)

returns true or false. Tests whether `x` is a wreath cycle. The argument `x` must be a wreath product element (see 2.2).

4.3.3 IsSparseWreathCycle

▷ `IsSparseWreathCycle(x)` (attribute)

returns true or false. Tests whether `x` is a sparse wreath cycle. The argument `x` must be a wreath product element (see 2.3).

4.3.4 WreathCycleDecomposition

▷ `WreathCycleDecomposition(x)` (attribute)

returns a list containing wreath cycles, namely the wreath cycle decomposition of `x`. The argument `x` must be a wreath product element (see 2.2).

4.3.5 Yade

▷ `Yade(x[, i])` (attribute)

returns a group element, namely the yade of the wreath cycle `x` evaluated at the smallest territory point. If the optional argument `i` is provided, the function returns the yade evaluated at the point `i`. The argument `x` must be a wreath cycle and the optional argument `i` must be an integer from the territory of `x` (see 2.3)

4.4 Printing, Viewing and Displaying

4.4.1 ViewObj (for a wreath product element)

▷ `ViewObj(x)` (method)
▷ `PrintObj(x)` (method)
▷ `Display(x[, optrec])` (method)

Wreath product elements are viewed, printed and displayed (see Section (**Reference: View and Print**) for the distinctions between these operations) as generic wreath product elements (see Section 2.1). For an example of the distinctions and outputs see 3.2.

The method `Display` allows an optional argument *optrec* which must be a record and modifies the display output for the execution of a single instance of the command.

For modifying the display output globally for all subsequent executions of `Display` see `SetDisplayOptionsForWreathProductElements` (4.4.3).

The following components of *optrec* are supported. Note, that in the following *labels* refer to the the printing output "$1, /ldots, m$" and "top" as seen in the tutorials.

`horizontal`
: `true` to use the horizontal printer. *DEFAULT*

  `false` to use the vertical printer.

`labels`
: `true` to print labels. *DEFAULT*

  `false` to suppress labels.

`labelStyle`
: `"none"` for labels in normal intensity. *DEFAULT*

  `"bold"` for labels in increased intensity.

  `"faint"` for labels in decreased intensity.

`labelColor`
: `"default"` for labels in the default GAP output color. *DEFAULT*

  `"red"` for labels in red color.

  `"blue"` for labels in blue color.

4.4.2 DisplayOptionsForWreathProductElements

▷ `DisplayOptionsForWreathProductElements()` (function)

prints the current global display options for wreath product elements.

4.4.3 SetDisplayOptionsForWreathProductElements

▷ `SetDisplayOptionsForWreathProductElements(`*optrec*`)` (function)

sets the current global display options for wreath product elements.

The argument *optrec* must be a record with components that are valid display options. (see 4.4) The components for the current global display options are set to the values specified by the components in *optrec*.

4.4.4 ResetDisplayOptionsForWreathProductElements

▷ `ResetDisplayOptionsForWreathProductElements()` (function)

resets the current global display options for wreath product elements to default.

4.5 Cycle Index of Wreath Products

The following functions construct the cycle index polynomial of wreath products in certain actions.

4.5.1 CycleIndexWreathProductImprimitiveAction

▷ `CycleIndexWreathProductImprimitiveAction(K, H)` (function)

For two permutation groups K and H this function constructs the cycle index polynomial of the wreath product $K \wr H$ in imprimitive action.

The implementation is based on [Pó37].

4.5.2 CycleIndexWreathProductProductAction

▷ `CycleIndexWreathProductProductAction(K, H)` (function)

For two permutation groups K and H this function constructs the cycle index polynomial of the wreath product $K \wr H$ in product action.

The implementation is based on [HH68] and [PR73].

Chapter 5

Operations

The generic representation of wreath product elements in wreath products of finite groups and in particular their (sparse) wreath cycle decompositions can be used to speed up certain computations in wreath products.

In particular this package provides efficient methods for finding conjugating elements, conjugacy classes, and centralisers. The implementations are based on [BNRW22] and references therein.

5.1 Operations List

Here we include a list of operations that take advantage of the generic representation of wreath product elements.

We include python scripts in the `dev/` directory that benchmark the WPE and native GAP implementations of these operations separately. The comparison of the runtimes supports the conclusion that the WPE implementations are an order of magnitude faster than the native GAP implementations. We can now solve these computational tasks for large wreath products that were previously not feasible in GAP

5.1.1 Wreath Product Representations

In the following let $G = K \wr H$ be a wreath product, where $H \leq \mathrm{Sym}(m)$.

In GAP the wreath product G can be given in one of the following representations :

- Generic Representation
- Permutation Representation in Imprimitive Action
- Permutation Representation in Product Action
- Matrix Representation

5.1.2 Operations for all Representations

Further let $x, y \in P = K \wr \mathrm{Sym}(m)$ be elements of the parent wreath product P which is given in the same representation as G.

The following operations use implementations that exploit the generic representation and (sparse) wreath cycle decompositions :

- `RepresentativeAction(G, x, y)`
- `Centraliser(G, x)`
- `ConjugacyClasses(G)`

5.1.3 Operations for Permutation Representations

Here we assume that G is given in some permutation representation.

The following operations use implementations that exploit the generic representation and (sparse) wreath cycle decompositions :

- `CycleIndex(G)`

5.1.4 Operations for Generic Representation

Here we assume that G is given in generic representation.

The following operations use implementations that exploit the generic representation and (sparse) wreath cycle decompositions :

- `Order(x)`

References

[BNRW22] Dominik Bernhardt, Alice C. Niemeyer, Friedrich Rober, and Lucas Wollenhaupt. Conjugacy classes and centralisers in wreath products. *J. Symbolic Comput.*, 113:97–125, 2022. 3, 7, 8, 23

[HH68] Michael A. Harrison and Robert G. High. On the cycle index of a product of permutation groups. *J. Combinatorial Theory*, 4:277–299, 1968. 22

[Ore42] Oystein Ore. Theory of monomial groups. *Trans. Amer. Math. Soc.*, 51:15–64, 1942. 3

[PÓ37] G. Pólya. Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen. *Acta Math.*, 68(1):145–254, 1937. 22

[PR73] E. M. Palmer and R. W. Robinson. Enumeration under two representations of the wreath product. *Acta Math.*, 131:123–143, 1973. 22

[Spe32] Wilhelm Specht. *Eine Verallgemeinerung der symmetrischen Gruppe*. PhD thesis, Humboldt-Universität zu Berlin, 1932. 3

Index

BaseComponentOfWreathProductElement, 19
BaseGroupOfWreathProduct, 19

ComponentsOfWreathProduct, 19
CycleIndexWreathProductImprimitive-
Action, 22
CycleIndexWreathProductProductAction,
22

Display
for a wreath product element, 20
DisplayOptionsForWreathProduct-
Elements, 21

IsomorphismWreathProduct, 18
IsSparseWreathCycle, 20
IsWreathCycle, 20

PrintObj
for a wreath product element, 20

ResetDisplayOptionsForWreathProduct-
Elements, 21

SetDisplayOptionsForWreathProduct-
Elements, 21

Territory, 19
TopComponentOfWreathProductElement, 19
TopGroupOfWreathProduct, 19

ViewObj
for a wreath product element, 20

WreathCycleDecomposition, 20

Yade, 20