COMPUTATIONAL GROUP THEORY

By

MICHAEL PATRICK MCALARNEN MARTIN

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree
With Honors in

Mathematics

THE UNIVERSITY OF ARIZONA

MAY 2015

Approved by:

Dr. Klaus Lux
Department of Mathematics

Abstract

The study of finite groups has been the subject of much research, with substantial success in the 20th century, in part due to the development of representation theory. Representation theory allows groups to be studied using the well-understood properties of linear algebra, however it requires the researcher to supply a representation of the group. One way to produce representations of groups is to take a representation of a subgroup and use it to induce a representation. We focus on the finite simple groups because they are the buliding blocks of an arbitrary simple group. This thesis investigates an algorithm to induce representations of large finite simple groups from a representation of a subgroup.

1 Introduction

One of the goals of group theory has been to study the properties of all of the finite simple groups. In particular, since properties of a group can be sometimes determined by their normal subgroups, this problem reduces to studying all of the finite simple groups. For example, Here, we shall study simple groups using their matrix representations, which allows for the use of the well-understood concepts of linear algebra. However, the simple groups we will deal with are very large — the Fischer group Fi_{23} has order about 4×10^{18} — and thus it is important to optimize calculations to reduce the number of matrix multiplications, which are quite slow when dealing with groups of this size.

1.1 GAP and Orb

The algorithms we are interested are implemented in a group algebra system called GAP[1] (Groups, Algorithms, and Programming), which is capable of performing operations on algebraic objects, which in our case are groups. For example, one may ask if a specific Klein 4-subgroup of S_4 is normal:

```
gap> s4 := SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> v4 := Group([(1,2)(3,4),(1,3)(2,4),(1,4)(2,3)]);
Group([ (1,2)(3,4), (1,3)(2,4), (1,4)(2,3) ])
gap> IsNormal(s4,v4);
true
```

which it is.

As the algorithm for inducing representations requires the orbits and stabilizers of elements under a group action, we make heavy use of the Orb package for GAP[2], which provides more functionality in regards to the handling of orbits. In particular, this package provides an Orb object that stores relevant information about an orbit of an element under a group action, such as the generators of that group and the Schreier tree of the orbit (see section 2.3). The Orb object is also used in several useful methods related to Schreier trees.

1.2 Representation Theory

In order to help understand the structure of groups, we can represent group elements as matrices. We call a representation of a group G a homomorphism

$$\varphi : G \to GL_n(F)$$

from the group into the group of invertible $n \times n$ matrices over a field F. If our group is simple, then this map is necessarily injective (if the homomorphism is nontrivial), and we can then restrict this representation to its image, yielding an isomorphism between the group and its matrix representation.

As an example, consider the group S_3, the symmetric group on three elements. In order to construct a matrix representation, we consider each permutation σ as a map on $\mathbb{R}^3$ by permuting the

[1] http://www.gap-system.org/
[2] http://gap-system.github.io/orb/

coordinates, via

$$\sigma(x_1, x_2, x_3) = (x_{\sigma(1)}, x_{\sigma(2)}, x_{\sigma(3)})$$

It is easy to check that this is a linear map, and thus we can consider its matrix over the standard basis. If, for example, $\mu = (1\ 2)$, then its matrix, M_μ, is

$$M_\mu = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

We can double-check that this representation is in fact a homomorphism by checking the presentation of S_3:

$$S_3 = \langle \rho, \mu | \rho^3 = \iota = \mu^2,\ \rho\mu\rho = \mu \rangle$$

Immediately, we see that $M_\mu^2 = \iota$, the 3×3 identity matrix, and now we check the other properties:

$$M_\rho = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$$M_\rho^3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

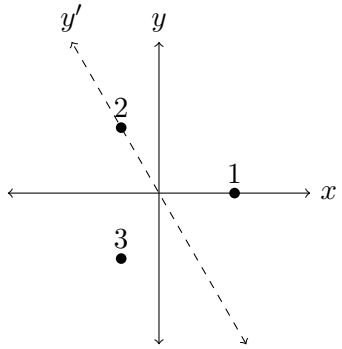
$$M_\rho M_\mu M_\rho = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

One important consideration when constructing these matrix representations is the dimension of the resulting matrices. Because we want to analyze these groups computationally, the smaller a matrix representation we can use, the faster the program will run. This isn't a big problem with this toy problem of S_3, however we want to eventually analyze very large groups, such as the aforementioned Fischer group Fi_{23} with about 4×10^{18} elements. One method of recognizing that a representation is larger than it needs to be is if there is a change of basis that results in the representation having an identity block. If such a basis exists,then the representation is said to be "reducible"; otherwise it is irreducible. In fact, this representation of S_3 is indeed reducible over the basis $\{x_1 + x_2 + x_3, x_1, x_2\}$, for which the representations of the generators are

$$(*) \qquad M_\rho = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & -1 \\ 0 & 1 & -1 \end{pmatrix} \qquad M_\mu = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

which has the 1×1 identity block in the upper left corner for both generators. In fact, we can actually use a 2×2 matrix representation for the elements of S_3, by choosing a smart basis of $\mathbb{R}^2$.

Recall that S_3 can be thought of as the motions (distance preserving maps) that preserve an equilateral triangle in the plane centered at the origin, we can choose our basis to be two vectors representing two of the vertices of the triangle, $\{x, y'\}$.



Using this basis, our generators become

$$M_\rho = \begin{pmatrix} 0 & -1 \\ 1 & -1 \end{pmatrix} \qquad M_\mu = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$

which can be seen to fulfill the properties of S_3 as well. Further, notice that these representations of the generators are precisely the lower right 2×2 block of their respective matrix representations in $(*)$, and our basis vectors are the second and third basis vectors of that matrix representation. In general, if a representation is reducible, then the map induced by removing the superfluous basis elements (the ones that result in the identity block) from the reducible representation is a representation as well.

Given a representation φ of a subgroup $H \leq G$, it is also possible to construct a representation called the "induced representation"

$$\varphi^G : G \to GL_m(F)$$

where $m = n \cdot [G : H]$. The key idea is that each element y of G permutes the cosets of H, which we can then use to produce a matrix of y.

To produce the matrices, we first recognize a very inefficient way to compute the matrix of an element x in a group G: by associating the i^{th} row and column with an element g_i of G, and producing the matrix of an element x by placing a 1 in the cell (i, j) where $x \cdot g_i = g_j$, yielding a $|G|$-dimensional matrix representation. For the induced representation, we will associate the i^{th} block in the row and column with a right coset of H in G, with coset representative r_i, and placing the matrix of $g \in H$, where

$$xr_i = r_j g$$

in the block (i, j). In essence, this is the same concept as the inefficient method of producing matrices, but with cosets instead of the elements themselves, and produces a $[G : H] \cdot m$-dimensional representation, where m is the dimension of the representation of H.

As we have already described an irreducible representation for S_3 above, to illustrate this concept, we will compute the matrix of $(1\ 2\ 3\ 4) \in S_4$ in the induced representation. First, we need to choose coset representatives for the cosets of S_3 in S_4:

$$r_1 = S_3(1\ 4) \qquad r_2 = S_3(2\ 4)$$
$$r_3 = S_3(3\ 4) \qquad r_4 = S_3 e$$

Now, because there are 4 cosets, and our representation of S_3 is 2-dimensional (mapping into $GL_2(\mathbb{R})$), the induced representation will be 8×8, consisting of blocks representing how the coset representatives act on (1 2 3 4). Now,

$$(1\ 2\ 3\ 4) \cdot r_i$$

and factor it into $g \cdot r_j$, where r_j is a coset representative and $g \in S_3$. Then, we fill in the i, j block with the representation of g. For example,

$$(1\ 2\ 3\ 4) \cdot \underbrace{(1\ 4)}_{r_1} = (1\ 2\ 3) = \underbrace{e}_{r_4} \cdot \underbrace{(1\ 2\ 3)}_{g}$$

and thus the block $(1, 4)$ is the representation of (1 2 3) in S_3. Repeating this process, we have

$$\begin{aligned}(1\ 2\ 3\ 4)(2\ 4) = (1\ 4)(2\ 3) &= (1\ 4) \cdot (2\ 3)\\ (1\ 2\ 3\ 4)(3\ 4) = (1\ 2\ 4) \quad &= (2\ 4) \cdot (1\ 2)\\ (1\ 2\ 3\ 4)e = (1\ 2\ 3\ 4) \ &= (3\ 4) \cdot (1\ 2\ 3)\end{aligned}$$

After computing the matrices of $(1\ 2\ 3), (2\ 3), (1\ 2) \in S_3$ and placing them in the appropriate spots in the matrix, we get the following representation for (1 2 3 4), where the blocks corresponding to the representations in S_3 have been bolded:

$$\begin{pmatrix} 0 & 0 & \mathbf{-1} & \mathbf{0} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{-1} & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{0} & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & \mathbf{-1} \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{-1} \\ \mathbf{0} & \mathbf{-1} & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{1} & \mathbf{-1} & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Further, the cyclic nature of (1 2 3 4) is visible in the position of the blocks.

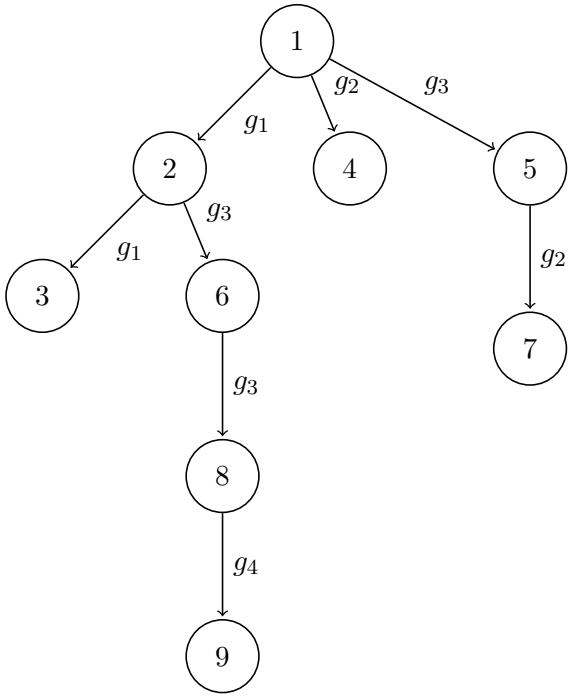
1.3 Computing the Induced Representation

Of course, we would like to be able to write a program that can compute the induced representation by itself, without our having to perform the multiplications and factorizations ourselves. It is fairly simple to write a program that produces the induced representation of S_4 from S_3 (sample code provided in the Appendix 6.1); however the tricky part is taking an arbitrary element h of S_3 and writing it in terms of the generators of S_3 so its matrix representation $\varphi(h)$ can be computed. In my version of the code, we take advantage of the fact that every element of S_3 can be written as $\mu^a \rho^b$, and an element has order 2 if and only if it has a factor of μ. These assumptions do not hold for an arbitrary group, of course, and thus we need a more general way of writing an element in terms of the generators of a group.

The process we use is called sifting, and it works by incrementally reducing the action of an element on a set through multiplication of generators, until the inverse of the element is found in terms of the generators. Once the inverse is found, the original element can be found easily.

In order to explain this process, we'll try to factor $x = (1\ 8\ 7\ 2\ 5\ 3)(4\ 6)$ in the subgroup $G < S_9$ generated by $\langle g_1, g_2, g_3, g_4 \rangle = \langle (1\ 2\ 3)(4\ 5), (1\ 4\ 5\ 7), (1\ 5)(2\ 6\ 8\ 4), (8\ 9) \rangle$. We want to find some word w in the generators such that the product $w^{-1}x$ fixes enough points that the only element of G that fixes those points is the identity. Thus, since $w^{-1}x = e$, $x = w$, and so we have factored x into the generators. In this case G is a transitive subgroup of S_9, so we need to fix each point in $1 \dots 9$. Let's first start with 1.

We see that $x(1) = 8$, and thus we need a word w in the generators such that $w(8) = 1$. To do this, we make a Schreier tree. A Schreier tree is a data structure that represents the orbit of an element a as an n-ary tree, where n is the number of generators. The root of the tree is the base element a, and the children of each node are the images of that node under each generator, and we also store which generator represents each edge. For example, the Schreier tree for the orbit of 1 is



From the tree, we can see that $g_3 g_3 g_1(1) = 8$, and thus $(g_3 g_3 g_1)^{-1} x$ stabilizes 1. Let $g_3 g_3 g_1 = w_1$ be the first word.

Now, we repeat the process, trying to stabilize 2, except now we start with $w_1^{-1}x$, and we will compute a second word, w_2, such that $w_2^{-1} w_1^{-1} x$ stabilizes 1 and 2. However, we cannot use the Schreier tree with the generators of G, as a word in the generators that stabilizes 2 does not necessarily stabilize 1. Thus, we need to compute G_1, the stabilizer group of 1, which is done using the Schreier-Sims algorithm[3]. This gives us a set of generators for G_1, which are defined in terms of the generators for G. We then compute a Schreier tree for 2 in G_1, and compute w_2. Eventually, we will have that $w_1, \dots, w_9$, where w_i is and element of $G_{1,\dots,i-1}$ such that $w_i^{-1} \cdots w_1^{-1} x$ stabilizes $1, \dots, i$. Thus, $w_9^{-1} \cdots w_1^{-1} x$ stabilizes all of $1 \dots 9$, and thus we have a factorization in terms of the generators:

$$x = w_1 \cdots w_9$$

[3]Grove, L. C. (1997) Preliminaries, in Groups and Characters, John Wiley & Sons, Inc., Hoboken, NJ, USA. doi: 10.1002/9781118032688.ch1

There is no guarantee that this will be the shortest word; for instance, the shortest word for x is $x = g_1 g_3 g_2^3 g_3 g_2 g_3 g_2$, however the previous algorithm will yield a much longer word, because each w_i is a word in the generators of G_i, which are themselves words in the g_is. Because we will be evaluating this word via matrix multiplication, a somewhat slow process ($O(n^3)$), we want to get as close to the shortest word as possible. The computation for the shortest word of x was found using the Factorization method in GAP, which computes all the words and finds the shortest. This is fine for smallish groups—this group G has about 350,000 elements and the Factorization method took about half a second to run—however this is unfeasible for larger groups. The task of reducing the length of the word found can be thought of having two parts: reducing the length of the stabilizer chain (or equivalently, ordering the elements to stabilize such that $G_{x_i} = G_{x_{i-1}}$ as often as possible)), which then reduces the number of nontrivial w_i's; and reducing the size of each w_i, which can be done through making the Schreier tree shallower, by introducing more generators. The latter must be done with discretion, however, as introducing more generators means the matrix representation of these elements must be computed as well, which could counteract the speedup of reducing the depth of the Schreier tree.

2 Previous work

The initial state of the code relied heavily on two functions: GrandUnifiedOrbitEnumerator and PQMatricesIndCond, the first of which computed information about the orbits of a subgroup, while the second, amongst other things prepares a group for sifting.

2.1 GrandUnifiedOrbitEnumerator

The Grand Unified Orbit Enumerator takes as its inputs a set of generators for a group, a set of generators for a subgroup of that group, and an upper bound on the desired number of orbits to find. It then calculates the orbits of the subgroup and stores associated information with each orbit, namely

- An orbit representative
- A word in the generators that maps 1 to the orbit representative
- A Schreier tree for the orbit, with the orbit representative as the base point

In addition to the list of orbits, which tell which elements are contained in which orbit, it also has a lookup table of what orbit contains each element.

3 Optimizations

The optimizations implemented through this thesis were to update the package to use the Orb package and to reduce the length of the words produced by the sifting procedure. There are two main factors contributing to the length of words produced by the sifting procedure: the length of the stabilizer chain (the number of subwords) and the depth of each Schreier tree (the length of the subwords). Previously, the GrandUnifiedOrbitEnumerator stored its calculated orbits as sparse lists[4] and included a list of Schreier trees corresponding to the orbits. With the Orb package,

[4]GAP, like Perl and some other languages, allows for lists where not all of the earlier indices have been instantiated.

Schreier trees are an attribute of the Orb object, and the routine MakeSchreierTreeShallow works well to reduce the depth of the Schreier tree. I also wrote a program to try to reduce the depth of Schreier trees, however it required about 25% more generators to be added in order to reduce to the same level as the Orb routine (my program is provided in the appendix for reference).

To reduce the length of the stabilizer chain, we can choose the sequence of elements to stabilize strategically so that stabilizing one element also stabilizes several other elements. Through some examples, we found that choosing the sequence of elements to be orbit representatives of the largest orbits at each stage resulted in the shortest stabilizer chain: 37% shorter than the chain produced by the StabilizerChain function in GAP (genss package), and 55% shorter than the chain produced by choosing orbit representatives of the shortest nontrivial orbits.

4 Future Work

Futher speedups could be achieved by improving on the MakeSchreierTreeShallow method, possibly by counting the length-two words of base elements represented in the edges of the Schreier tree and repeatedly adding the most frequent word to the base. Implementing a more efficient matrix multiplication algorithm can also speed up the computation of the induced representation matrices.

5 Appendix: Assorted GAP code

5.1 Induced Representation of S_4

```
RepresentS4:=function(permutation)
        # Given an element of S4, this will return the
        # induced representation from S3
        local flip, rot, representation, lhs;
        # S3 is generated by (1 2), (1 2 3)
        # Representation of (1 2) is:
        #   | 0 1 |
        #   | 1 0 |
        #
        flip := [[0,1],[1,0]];
        # Representation of (1 2 3) is:
        #   | 0 -1 |
        #   | 1 -1 |
        rot  := [[0,-1],[1,-1]];

        # Representation of elements in S4 is an 8x8 matrix
        representation := TransposedMat([[0,0,0,0,0,0,0,0]])
                                * [[0,0,0,0,0,0,0,0]];

        # Cosets of S4 / S3 are (1,4); (2, 4); (3,4); e
        # Box (i,j) is the matrix of x such that
        # cosetrep_i * permutation = x * cosetrep_j
```

```
        # (1,4)
        representation := ComputeRep(permutation, 1,
                                representation, flip, rot);
        # (2,4)
        representation := ComputeRep(permutation, 2,
                                representation, flip, rot);
        # (3,4)
        representation := ComputeRep(permutation, 3,
                                representation, flip, rot);
        # (4,4) = e
        representation := ComputeRep(permutation, 4,
                                representation, flip, rot);

        return representation;
end;

ComputeRep:=function(permutation, left, representation,
                                flip, rotation)
        local lhs, box, basepart, factors, insert;
        # Given the representative (1, left) multiplied onto
        #  the left of the permutation, fill the appropriate
        # spot in the representation

        if (left = 4)
        then
                lhs := permutation;
        else
                lhs := (left,4) * permutation;
        fi;
        # figure out which coset this is in by looking at
        # where it sends 4
        box := (4) ^ lhs;

        # Figure out the S3 part
        if (box = 4)
        then
                basepart := lhs;
        else
                basepart := lhs * (box,4);
        fi;
        # Factor the basepart
        factors := FactorPerm(basepart);
        insert  := flip^factors[1] * rotation^factors[2];

        # Fill the appropriate spot in the representation
        representation[left*2-1][box*2-1] := insert[1][1];
        representation[left*2][box*2-1] := insert[2][1];
        representation[left*2-1][box*2] := insert[1][2];
```

```
        representation[left*2][box*2] := insert[2][2];

        return representation;
end;

FactorPerm:=function(permutation)
        local a,b;
        # Given a permutation in S3, returns [a, b] such that
        # permutation = (1,2)^a(1,2,3)^b
        # There is probably a better way to do this
        a:=0;
        b:=0;

        while (Order(permutation) = 2)
        do
        # Order(permutation) =2 iff a > 0
                permutation := (1,2) *permutation;
                a := a+1;
        od;
        while (Order(permutation) = 3)
        do
        # Order(permutation) = 3 iff b > 0 (now that
        #  there are no factors of (1,2) )
                permutation := (1,3,2) * permutation;
                b := b+1;
        od;
        return [a,b];
end;
```

5.2 Producing a Shallower Schreier Tree

5.2.1 Deterministic

```
# Takes one argument, an orbit object, and an optional integer
# argument to specify the desired depth of the Schreier tree.
# If the desired depth is not supplied, the log2 of the size of
# the orbit is used.
ShallowerTree:=function(arg)
        local orbit, depth, base_gens, to_add, prev, newgen, ct;

        orbit := arg[1];
        if (Length(arg) > 1) then
                depth := arg[2];
        else
                depth := Int(Log2(Float(Length(orbit))));
        fi;

        base_gens := [orbit!.gens];
        prev := base_gens;
```

```
        ct:=Length(base_gens[1]);

        while DepthOfSchreierTree(orbit) > depth do
                Print("Current depth: ", DepthOfSchreierTree(orbit),",
                        number of generators: ", ct,"\n");
                # Want to add all the 2-words, then the 3-words, and so on
                # Apply the generators to each of the previous generators
                to_add := Flat(TransposedMat(prev)* base_gens); # on the right
                to_add := Unique(to_add);
                for newgen in to_add do
                        ct:=ct+1;
                        orbit := AddGeneratorsToOrbit(orbit, [newgen]);
                        if DepthOfSchreierTree(orbit) < depth then
                                break;
                        fi;
                od;
                prev := [to_add];
        od;
        return orbit;
end;
```

5.2.2 Monte Carlo

```
ShallowerTreeRandom:=function(arg)
        local orbit, depth, base_gens, to_add, prev, newgen, ct;

        orbit := arg[1];
        if (Length(arg) > 1) then
                depth := arg[2];
        else
                depth := Int(Log2(Float(Length(orbit))));
        fi;

        base_gens := [orbit!.gens];
        prev := base_gens;

        ct:=Length(base_gens[1]);

        while DepthOfSchreierTree(orbit) > depth do
                Print("Current depth: ", DepthOfSchreierTree(orbit),",
                        number of generators: ", ct,"\n");
                # Want to add a random 2-word with the
                # most popular generators
                to_add := ();
                # First term
                to_add := to_add * orbit!.gens[
                                orbit!.schreiergen[ Random(2, Size(orbit))]];
```

```
                # Second term
                to_add := to_add * orbit!.gens[
                                orbit!.schreiergen[ Random(2, Size(orbit))]];
                # Make sure it is a new generator
                if to_add in orbit!.gens then continue; fi;

                orbit := AddGeneratorsToOrbit(orbit, [to_add]);
                ct := ct + 1;
        od;
        return ct;
end;
```

5.3 Computing a Stabilizer Chain Base

```
MaxOrbitEl:=function(orb)
  local lengths,max_index,i;

  lengths := List(orb, Length);
  max_index := 1;
  for i in [1..Length(lengths)] do
        if (lengths[i] > lengths[max_index]) then
                max_index := i;
        fi;
  od;
  return orb[max_index][1];

end;

StabChainFinder:=function(gp, set)
        local rep, stabgp, orbits, orb_rep, record, OrbitEl;


        record :=rec();
        record.base := [];
        record.stabilizers := [];
        stabgp := gp;

        while not IsTrivial(stabgp) do
                orbits := Orbits(stabgp, set);
                orb_rep := MaxOrbitEl(orbits);
                stabgp := Stabilizer(stabgp, orb_rep);
                Print("Orbit of ", orb_rep, " chosen, yielding group of size ",
                        Size(stabgp),"\n");
                Add(record.base, orb_rep);
                Add(record.stabilizers, stabgp);
        od;
        return record;
end;
```

References

"GAP System for Computational Discrete Algebra." GAP System for Computational Discrete Algebra. GAP Centers. Web. 5 May 2015. <http://www.gap-system.org/>.

Grove, L. C. (1997) Preliminaries, in Groups and Characters, John Wiley & Sons, Inc., Hoboken, NJ, USA. doi: 10.1002/9781118032688.ch1

Mueller, Juergen, Max Neunhöffer, and Felix Noeske. "Orb." GAP Package. Web. 5 May 2015. <http://gap-system.github.io/orb/>.