The Schreier-Sims algorithm for matrix groups

Henrik Bäärnhielm

URL: http://matrixss.sourceforge.net/
E-mail address: henrik.baarnhielm@imperial.ac.uk

Abstract. This is the report of a project with the aim to make a new implementation of the Schreier-Sims algorithm in GAP, specialized for matrix groups. The standard Schreier-Sims algorithm is described in some detail, followed by descriptions of the probabilistic Schreier-Sims algorithm and the Schreier-Todd-Coxeter-Sims algorithm. Then we discuss our implementation and some optimisations, and finally we report on the performance of our implementation, as compared to the existing implementation in GAP, and we give benchmark results. The conclusion is that our implementation in some cases is faster and consumes much less memory.

Contents

Preface

This report is made as part of a student project at the Department of Mathematics, Imperial College of Science, Technology and Medicine in London, United Kingdom. It is made in partial fulfillment of the requirements for the degree of Master of Science in Pure Mathematics.

I would like to thank my supervisors Prof. Alexander Ivanov at Imperial College and Dr. Leonard Soicher at Queen Mary, University of London, for their encouragement and help.

I am also particularly grateful to Alexander Hulpke at Colorado State University in Fort Collins, USA, for suggesting this project to me, and for his constant help and advice during my work.

CHAPTER 1

Introduction

The following report makes up one the two parts of a project in computational group theory, the other part being a software package for the computer system GAP (see [**GAP**]), which can be found on the WWW at the URL given on the title page. This text will describe the mathematics that provide the foundations of the package, including the algorithms used and their complexity, and also some of the more computer science oriented aspects, like what datastructures that were used, and how the implementation was done.

Computational group theory (CGT) is an area of research on the border between group theory and computer science, and work in CGT is often of both theoretical (mathematical) and practical (programming) nature, leading to both theoretical results (mathematical theorems and proofs) and practical results (software), and this project is no exception. Introductory surveys of CGT can be found in [**Sim03**], [**Ser97**], [**Neu95**] and [**CH92**].

The aim of the project was to make a GAP package with an implementation of the Schreier-Sims algorithm for matrix groups. The Schreier-Sims algorithm computes a base and a strong generating set for a group, and an implementation of this fundamental algorithm is already included in the standard GAP distribution, but that implementation always first computes a faithful action (ie a permutation representation) of the given group and then executes the algorithm on the resulting permutation group. The idea for this project was to restrict attention to matrix groups, and implement a version of the algorithm which works with the matrices directly, and see if one can obtain a more efficient implementation in this way.

A survey of computational matrix group theory can be found in [**NP01**]. It should be noted that we are only interested in finite groups, ie matrix groups over finite fields, and therefore there is no need to worry about any noncomputability or undecidability issues.

We will begin with a quick reference of the basic concepts from group theory and computer science that are being used, before moving on to describe the Schreier-Sims algorithm. The description will be quite detailed, and then we will turn to the variants of the algorithm that have also been implemented in the project: the random (ie. probabilistic) Schreier-Sims algorithm and the Schreier-Todd-Coxeter-Sims algorithm. After that we will say something about the implementation, and describe some tricks and improvements that have been done to make the algorithm faster. Finally, the practical performance and benchmark results of the implementation will be shown, and compared to the existing implementation in GAP.

It must be mentioned that a report similar to this one is [**Mur93**], from which a fair amount of inspiration comes.

CHAPTER 2

Preliminaries

The definitions and statements in this section are assumed to be known, but we state them anyway, since authors often use different notation and sometimes put slightly different meaning to some of the following concepts (eg. the exact definition of graphs tend to vary).

1. Graph theory

First some graph theory, where we follow [**Big89**].

DEFINITION 2.1. A *directed graph* is an ordered pair $G = (V, E)$ where V is a finite non-empty set, the *vertices* of G and $E \subseteq V \times V$ is the *edges* of G.

REMARK 2.2. A graph in the sense of 2.1 is sometimes called a *combinatorial graph* in the literature, to emphasize that they are not *metric graphs* in the sense of [**BH99**]. As we are interested only in finite graphs and not in geometry, we do not make use of this nomenclature.

REMARK 2.3. The definition implies that our graphs have no multiple edges, but may have loops. We will henceforth omit the word "directed" since these are the only graphs we are interested in.

DEFINITION 2.4. If $G = (V, E)$ is a graph, a sequence $v = v_1, v_2, \ldots, v_n = u$ of vertices of G such that $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, n-1$ is called a *walk* from v to u. The *length* of a walk $v = v_1, \ldots, v_k = u$ is $k - 1$. If $v = u$ then the walk is called a *cycle*. If all vertices in the walk are distinct, it is called a *path*. If every pair of distinct vertices in G can be joined by a path, then G is *connected*.

DEFINITION 2.5. A graph $G' = (V', E')$ is a *subgraph* of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

DEFINITION 2.6. A *tree* is a connected graph without cycles. A *rooted tree* is a tree $T = (V, E)$ where a vertex $r \in V$ have been specified as the *root*.

DEFINITION 2.7. If $G = (V, E)$ is a graph, then a tree T is a *spanning tree* of G if T is a subgraph of G and T have vertex set V.

The following are elementary and the proofs are omitted.

PROPOSITION 2.8. *In a tree $T = (V, E)$ we have $|E| = |V| - 1$ and there exists a* unique *path between every pair of distinct vertices. Conversely, if G is a graph where every pair of distinct vertices can be joined by a unique path, then G is a tree.*

PROPOSITION 2.9. *Every graph has a spanning tree.*

DEFINITION 2.10. Let $T = (V, E)$ be a rooted tree with root $r \in V$. The *depth* of a node $x \in V$ is the length of the path from x to r. The *height* of T is the maximum depth of any node.

DEFINITION 2.11. If $G = (V, E)$ is a graph, then a *labelling* of G is a function $w : E \to L$ where L is some set of "labels". A *labelled graph* is a graph with a corresponding labelling.

2. Group theory

Now some group theory, where we follow our standard references [**BB96**] and [**Ros94**].

NOTE 2.12. All groups in this report are assumed to be finite.

DEFINITION 2.13. If $G = \langle S \rangle$ is a group, then the *Cayley graph* $\mathcal{C}_G(S)$ is the graph with vertex set G and edges $E = \{(g, sg) \mid g \in G, s \in S\}$.

DEFINITION 2.14. An *action* of a group G on a finite set X is a homomorphism $\lambda : G \to \mathrm{Sym}(X)$ (where $\mathrm{Sym}(X)$ is the group of permutations on X). If λ is injective, the action is *faithful*.

REMARK 2.15. Following a convention in computational group theory, actions are from the right, and $\lambda(g)x$ is abbreviated with x^g, for $g \in G$ and $x \in X$. The elements of X are called *points*. Note that some of the rules for exponents hold since we have an action, eg $(x^g)^h = x^{gh}$.

DEFINITION 2.16. Let G be a group acting on the finite set X. For each point $\alpha \in X$, the *orbit* of α is $\alpha^G = \{\beta \in X \mid \beta = \alpha^g, g \in G\}$ and the *stabiliser* of α is $G_\alpha = \{g \in G \mid \alpha^g = \alpha\}$.

The following are elementary and the proofs are omitted.

PROPOSITION 2.17. *Let G be a group acting on the finite set X. For each $p \in X$ we have $G_p \leq G$, and so we can define inductively*

$$G_{\alpha_1,\alpha_2,\ldots,\alpha_n} = (G_{\alpha_1,\alpha_2,\ldots,\alpha_{n-1}})_{\alpha_n} \tag{2.1}$$

where $n > 1$ and $\alpha_1, \ldots, \alpha_n \in X$.

PROPOSITION 2.18. *Let G be a group acting on the finite set X. For each $p \in X$, the map $\mu_p : G/G_p \to p^G$ given by*

$$G_p g \mapsto p^g \tag{2.2}$$

for each $g \in G$, is a bijection. In particular, $\left|p^G\right| = [G : G_p]$.

3. Computer science

When it comes to computer science, our standard reference is [**CLR90**] where all basic computer science notions that are used here can be found. First of all, complexity analysis of algorithms and its asymptotic notation, in particular the $\mathrm{O}(\cdot)$-notation, is assumed to be known. Basic graph algorithms like breadth-first search, computation of connected compontents and spanning tree algorithms are also assumed to be known.

Hash tables will likewise be used without further explanation. Even though the code in the project does not use hashing explicitly, but rely on GAP for that, it is worth mentioning that, to the author's knowledge, the best general-purpose hash function known to humanity is described in [**Jen97**].

CHAPTER 3

The Schreier-Sims algorithm

We will now describe the Schreier-Sims algorithm, and the references we are using are [**CSS99**], [**Ser03**] and [**But91**]. First we have to describe some background and clarify the problem that the algorithm solves.

1. Background and motivation

The overall goal for work in group theory is to understand groups, and to answer various questions about groups. In particular, in CGT the interest is focused on questions of algorithmic nature, so given a group G we are interested in algorithms for things like the following:

- What is $|G|$?
- List all elements of G, without repetition.
- If $G \leq H$ and we are given an arbitrary $g \in H$, is it true that $g \in G$? This is referred to as *the membership problem.*
- Generate a random $g \in G$.
- Is G abelian (soluble, polycyclic, nilpotent)?
- Find representatives for the conjugacy classes of G.
- Find a composition series for G, including the (isomorphism classes of the) composition factors.
- Given $g \in G$ or $H \leq G$ find the centralizer of g or the normalizer of H, respectively.

To accomplish these tasks we need a computer representation of G, ie a datastructure. A common way in which a group is given is via a generating set, but this alone does not help in solving our problems, so we need a better representation. It must be possible to compute this representation from a generating set, and a nice property would be if subgroups of G in some direct way inherit the representation, so that divide-and-conquer techniques could be used when designing algorithms (see [**CLR90**]).

2. Base and strong generating set

Consider the situation where we have a chain of subgroups of G

$$G = G^0 \geq G^1 \geq \cdots \geq G^n = 1 \tag{2.1}$$

Each $g \in G$ can be written $g = g_1 u_1$ where u_1 is a representative of $G^1 g$ and $g_1 \in G^1$, and inductively we can factorize g as $g = u_n u_{n-1} \cdots u_1$, since the subgroup chain reaches 1. Moreover, this factorization is *unique* since for each i the cosets of G^{i+1} partition G^i, and by the same reason, different group elements will have different factorizations.

We thus see that if we know generating sets for the subgroups in such a subgroup chain, and if we know a right transversal T^i of the cosets of G^{i+1} in G^i for each $i = 0, \ldots, n-1$, then we could easily solve at least the first two listed problems listed in section 1. By Lagrange we know $|G| = \left|T^1\right| \left|G^1\right|$ and inductively $|G| = \left|T^1\right| \cdots |T^n|$, which solves the first problem. Using the factorization we have a bijection from G to $T^1 \times T^2 \times \cdots \times T^n$ so by enumerating elements of the latter set and multiplying in G we can list the elements of G without repetition.

Now we introduce a special type of subgroup chain.

DEFINITION 3.1. Let G be a group acting on the finite set X. A sequence of points $(\alpha_1, \ldots, \alpha_n)$ of X such that $G_{\alpha_1,\ldots,\alpha_n} = 1$ is called a *base* for G. Note that the base determines a *stabiliser chain*

$$G \geq G_{\alpha_1} \geq \cdots \geq G_{\alpha_1,\ldots,\alpha_n} = 1 \tag{2.2}$$

Let $G^i = G_{\alpha_1,\ldots,\alpha_i}$ for $i = 1, \ldots, n$. A generating set S for G such that $\left\langle S \cap G^i \right\rangle = G^i$ for all i is called a *strong generating set* (SGS) for G.

The concept of a base and strong generating was first introduced in [**Sim70**] in the context of permutation groups, and is of fundamental importance, though mainly for permutation groups. We already know that the first two of our problems can be solved if we know a subgroup chain and we shall see that with a base and strong generating set the membership problem can also easily be solved. The Schreier-Sims algorithm computes a base and strong generating set for a group G given a generating set S, and since it is an efficient algorithm if G is a permutation group, the concept of base and strong generating set has become very important. Many more sophisticated algorithms for permutation groups require a base and strong generating set as input. For matrix groups, on the other hand, the situation is a bit more complicated, as we shall see later.

3. Schreier trees

Before giving the Schreier-Sims algorithm itself, there are a few auxiliary algorithms that must be explained. Consider therefore a group $G = \langle S \rangle$ that acts on a finite set X. From the previous section we know that even if we have a base $(\alpha_1, \ldots, \alpha_n)$ for G, with corresponding stabiliser chain $G \geq G^1 \geq \cdots G^n = 1$, we also need to find right transversals of the cosets of G^{i+1} in G^i for $i = 1, \ldots, n-1$. But since these groups are stabilisers from the action on X, we can use Proposition 2.18 and instead find the orbits $\alpha_1^G, \alpha_2^{G^1}, \ldots, \alpha_n^{G^{n-1}}$. We will see that the orbits are straightforward to compute.

The action of G on X can be represented by a labelled graph with labels from S, analogous to the Cayley graph $\mathcal{C}_S(G)$ of G from Definition 2.13. Let the vertices of the graph be X and the edges be $\{(p, p^g) \mid p \in X, g \in G\}$, where the edge (p, p^g) is labelled by g. Obviously the orbits of the action are the connected components of this graph, so α_1^G is the component containing α_1. We are interested in finding this orbit and to store it in a datatstructure, and we are thus led to consider a spanning tree of the connected component, since such a tree contains enough information for us. The edges of the graph that are left out do not give us any additional relevant information about the action of G, only alternative ways to move between the points, and trees are considerably easier to store than graphs.

DEFINITION 3.2. Let G be a group acting on the finite set X and let $\alpha \in X$. A spanning tree rooted at α for the component in the corresponding graph containing α is called a *Schreier tree* for the orbit α^G.

The Schreier tree can be computed by a simple breadth-first search of the component containing α, and thus we have an algorithm for finding the orbits. However, it is of course not a computationally good idea to explictly generate the graph, then compute the connected components and finally find the Schreier trees. A breadth-first search to find the Schreier tree can be done without the graph itself, as demonstrated by Algorithm 3.1.

Algorithm 3.1: `ComputeSchreierTree`

Data: A group $G = \langle S \rangle$ acting on a finite set X and a point $\alpha \in X$.
Result: A Schreier tree for α^G.
/*Assumes the existence of a function `Tree`(x) that creates an empty tree with root x and a function `AddChild`(T, p_1, p_2, l) that adds p_2 as a child to p_1 in the tree T, with label l. */

```
1  begin
2      points := {α}
3      tree := Tree(α)
4      repeat
5          children := ∅
6          foreach p ∈ points do
7              foreach s ∈ S do
8                  p' := p^s
9                  if p' ∉ tree then
10                     AddChild(tree, p, p', s)
11                     children := children ∪ {p'}
12                 end
13             end
14         end
15         points := children
16     until points = ∅
17     return tree
18 end
```

We noted above that we used Proposition 2.18 to store the orbits instead of the coset representatives, but we still need the latter, so we must be able to compute them. Fortunately, that is straightforward to do when we have a Schreier tree. Assume we have a Schreier tree $T = (V, E)$ for the orbit α^G for some point $\alpha \in X$. If $g \in G$ then $\alpha^g \in V$ and there is a path from α to α^g in T. Let $s_1, s_2, \ldots, s_n$ be the labels of the path, so that $s_i \in S$ for $i = 1, \ldots, n$, and let $h = s_1 s_2 \cdots s_n$. Then obviously $\alpha^h = \alpha^g$ so $G_\alpha g = G_\alpha h$ and h is a coset representative for the coset of g. Moreover, h is unique since T is a tree. Thus, to find the coset representative for g we only have to follow the unique path from α^g to the root α and multiply the edge labels. Algorithm 3.2 performs the slightly more general task of following the path from a given point to the root.

Algorithm 3.2: `OrbitElement`

Data: A group $G = \langle S \rangle$ acting on a finite set X, a Schreier tree T for the orbit α^G of the point $\alpha \in X$ and an arbitrary point $p \in X$.
Result: The element $g \in G$ such that $\alpha^g = p$
/*Assumes the existence of a function `EdgeLabel`(T, p) that returns the label of the unique edge between p and its parent in T */

```
1 begin
2     g := 1
3     while p ≠ α do
4         s := EdgeLabel(T, p)
5         p := p^(s^-1)
6         g := sg
7     end
8     return g
9 end
```

We defer the complexity analysis of these algorithms until later.

4. Formulating the problem

To more formally state the problem solved by the Schreier-Sims algorithm, the following is needed.

Definition 3.3. Let G be a group acting on the finite set X. A sequence of points $B = (\alpha_1, \ldots, \alpha_n)$ of X and a generating set S for G, such that no element of S fixes all points of B, is called a *partial base* and *partial strong generating set*, respectively.

Remark 3.4. A base and strong generating set as in Definition 3.2 are called *complete*.

Remark 3.5. If we define $G^i = G_{\alpha_1,\ldots,\alpha_i}$, $S^i = S \cap G^i$ and $H^i = \langle S^i \rangle$ for $i = 1, \ldots, n$, we see that $H^n = 1$ since no element of S fixes all points of B (and we use the convention that $\langle \emptyset \rangle = 1$). We therefore have

$$G \geq G^1 \geq \cdots \geq G^n \tag{4.1}$$

$$G \geq H^1 \geq \cdots \geq H^n = 1 \tag{4.2}$$

Moreover, $G^i \geq \langle S^i \rangle = H^i$ for $i = 1, \ldots, n$ and if we have equality then, by Definition 3.2, S and B are complete. If $h \in H^{i+1} = \langle S^{i+1} \rangle = \langle S \cap G^{i+1} \rangle$ then $h = s_1 \cdots s_k$ where $\alpha_{i+1}^{s_j} = \alpha_{i+1}$ for $j = 1, \ldots, k$ so $\alpha_{i+1}^h = \alpha_{i+1}$ and therefore $h \in H^i_{\alpha_{i+1}}$. Thus $H^{i+1} \leq H^i_{\alpha_{i+1}}$ for $i = 0, \ldots, n-1$.

Now our problem can be stated as follows: given a group G acting on the finite set X, together with a partial base B with points from X and partial strong generating set S, either verify that B is a (complete) base and that S is a (complete) strong generating set, or extend B and S so that they become complete. This is the problem that is solved by the Schreier-Sims algorithm.

The following result from [**Leo80**] is used when designing the algorithm.

THEOREM 3.6. *Let G be a group acting on the finite set X, and let $B = (\alpha_1, \ldots, \alpha_n)$ be a partial base and S a partial strong generating set for G. Let also $G^i = G_{\alpha_1,\ldots,\alpha_i}$, $S^i = S \cap G^i$, $H^i = \langle S^i \rangle$ for $i = 1, \ldots, n$ and $G = G^0 = H^0$. Then the following statements are equivalent:*

1. *B and S are complete.*
2. *$G^i = H^i$ for $i = 0, \ldots, n$.*
3. *$H^i_{\alpha_{i+1}} = H^{i+1}$ for $i = 0, \ldots, n-1$.*
4. *$[H^i : H^{i+1}] = \left|\alpha_{i+1}^{H^i}\right|$ for $i = 0, \ldots, n-1$.*

PROOF. From Remark 3.5 we know that (1) and (2) are equivalent. Assuming (2) we have

$$H^i_{\alpha_{i+1}} = G^i_{\alpha_{i+1}} = G^{i+1} = H^{i+1} \tag{4.3}$$

for $i = 0, \ldots, n-1$ which is precisely (3). If we instead assume (3) and also assume for induction that $G^i = H^i$ (the base case $G = H^0 = G^0$ is ok) then

$$G^{i+1} = G^i_{\alpha_{i+1}} = H^i_{\alpha_{i+1}} = H^{i+1} \tag{4.4}$$

so by induction we get $G^i = H^i$ for $i = 0, \ldots, n$, which is (2).

Now assume (3) and note that from 2.18 we have $[H^i : H^i_{\alpha_{i+1}}] = \left|\alpha_{i+1}^{H^i}\right|$, so since $H^i_{\alpha_{i+1}} = H^{i+1}$ we get (4). Finally, assume (4). From Remark 3.5 we know $H^i_{\alpha_{i+1}} \geq H^{i+1}$ so if we again use 2.18 we get $\left|\alpha_{i+1}^{H^i}\right| = [H^i : H^{i+1}] \geq [H^i : H^i_{\alpha_{i+1}}] = \left|\alpha_{i+1}^{H^i}\right|$ and thus $H^i_{\alpha_{i+1}} = H^{i+1}$. □

As observed earlier, we are often given a group in the form of a generating set, but Schreier-Sims algorithm requires a partial base and a partial strong generating set as input. Those are easy to compute, though, using Algorithm 3.3. The algorithm also makes sure that the partial strong generating set is closed under inverses and does not contain the identity, which removes the need to consider some special cases later on. We will see how the function `NewBasePoint` that is used in Algorithm 3.3 can be implemented when G is a matrix group.

5. Schreier's Lemma

The name Schreier in Schreier-Sims algorithm comes from the following result, which in our case allows us to find a generating set for a stabiliser. It first appeared in [**Sch27**], and our proof is originally from [**Hal59**].

THEOREM 3.7 (Schreier's Lemma). *Let $G = \langle S \rangle$ be a group, let $H \leq G$ and let T be a right transversal of the cosets of H in G. For $g \in G$, let $\bar{g} \in T$ be the unique element such that $Hg = H\bar{g}$. Then H is generated by*

$$S_H = \left\{ ts(\overline{ts})^{-1} \mid t \in T, s \in S \right\} \tag{5.1}$$

PROOF. Without loss of generality we can assume that $1 \in T$ (the coset representative of H itself). By definition, $Hts = H\overline{ts}$ which implies that $ts(\overline{ts})^{-1} \in H$ for all $t \in T, s \in S$. Hence, $S_H \subseteq H$ and $\langle S_H \rangle \leq H$, so the content of the statement lies in the other inclusion.

Let $h \in H \leq G$ and observe that since $\langle S \rangle = G$ we have $h = s_1 s_2 \cdots s_k$ where $s_i \in S \cup S^{-1}$ for $i = 1, \ldots, k$. Define a sequence $t_1, t_2, \ldots, t_{k+1}$ of $k+1$ elements of

Algorithm 3.3: `GetPartialBSGS`

Data: A group $G = \langle S \rangle$ acting on a finite set X and a sequence of points B of X (possibly empty).
Result: A partial base B' and partial strong generating set S' for G.
/*Assumes the existence of a function `NewBasePoint`(g) that returns a point $p \in X$ such that $p^g \neq p$ */

```
1  begin
2      base := B
3      sgs := ∅
4      foreach s ∈ S \ {1} do
5          if base^s = base then
6              point := NewBasePoint(s)
7              base := base ∪ {point}
8          end
9          sgs := sgs ∪ {s, s^{-1}}
10     end
11     return (base, sgs)
12 end
```

T as follows: $t_1 = 1$ and inductively $t_{i+1} = \overline{t_i s_i}$. Furthermore, let $a_i = t_i s_i t_{i+1}^{-1}$ for $i = 1, \ldots, n$ and observe that

$$h = (t_1 s_1 t_2^{-1})(t_2 s_2 t_3^{-1}) \cdots (t_n s_n t_{n+1}^{-1}) t_{n+1} = a_1 a_2 \cdots a_n t_{n+1} \tag{5.2}$$

We now show that $a_i \in \langle S_H \rangle$ for $i = 1, \ldots, n$, and that $t_{n+1} = 1$, which implies that $H \leq \langle S_H \rangle$.

For each $i = 1, \ldots, n$, either $s_i \in S$ or $s_i^{-1} \in S$. In the first case we immediately get $a_i = t_i s_i (\overline{t_i s_i})^{-1} \in S_H$, and in the second case we have $H t_{i+1} s_i^{-1} = H \overline{t_i s_i} s_i^{-1} = H t_i s_i s_i^{-1} = H t_i$ which implies that $t_i = \overline{t_{i+1} s_i^{-1}}$. Hence, $a_i^{-1} = t_{i+1} s_i^{-1} t_i^{-1} = t_{i+1} s_i^{-1} \left(\overline{t_{i+1} s_i^{-1}}\right)^{-1} \in S_H$ and thus $a_i \in \langle S_H \rangle$.

Finally, since $h \in H$ and $\langle S_H \rangle \leq H$ we have $t_{n+1} = (a_1 a_2 \cdots a_n)^{-1} h \in H$, so t_{n+1} is the coset representative of H, and therefore $t_{n+1} = 1$. Thus $H \leq \langle S_H \rangle$. □

Our situtation is that we have a group $G = \langle S \rangle$ acting on the finite set X, and we want to use Schreier's Lemma to find the generators (usually called the *Schreier generators*) for the stabiliser G_α, where $\alpha \in X$. If we compute a Schreier tree for the orbit α^G using Algorithm 3.1 then we know that Algorithm 3.2 can be used to find the transversal of the cosets of G_α in G.

For $p \in X$, let $t(p) \in G$ denote the result of Algorithm 3.2. Using the notation in Theorem 3.7 we then have $\bar{g} = t(\alpha^g)$ and the transversal is $\{t(p) \mid p \in \alpha^G\}$, so for $s \in S$ and $p \in \alpha^G$ the Schreier generator can be expressed as

$$t(p) s t(\alpha^{t(p)s})^{-1} = t(p) s t((\alpha^{t(p)})^s)^{-1} = t(p) s t(p^s)^{-1} \tag{5.3}$$

and a generating set for G_α is

$$\{t(p) s t(p^s)^{-1} \mid p \in \alpha^G, s \in S\} \tag{5.4}$$

5.1. Computing a base and SGS. Going back to our problem, if we have a partial base $B = (\alpha_1, \ldots, \alpha_n)$ for G with corresponding partial strong generating set S, then we can use Schreier's Lemma to solve our problem. Using the notation from Theorem 3.6, we calculate the Schreier generators for each G^i, using (5.4) and add them to S, possibly adding points to B, if some Schreier generator fixes the whole base, in order to ensure that B and S are still partial. When this is finished, we have $H^i = G^i$ for $i = 1, \ldots, n$ and thus B and S are complete. We can use Algorithm 3.4 to calculate a Schreier generator.

Algorithm 3.4: `GetSchreierGenerator`

Data: A group $G = \langle S \rangle$ acting on a finite set X, a Schreier tree T for the orbit α^G of the point $\alpha \in X$, a $p \in X$ and a generator $s \in S$.
Result: The Schreier generator corresponding to p and s

```
1 begin
2     t_1 := OrbitElement(T, p)
3     t_2 := OrbitElement(T, p^s)
4     return t_1 s t_2
5 end
```

However, there is a problem with this simple approach, in that the generating sets defined in (5.4) can be very large, and contain many redundant generators. A fraction of the Schreier generators is usually enough to generate the stabiliser. In [**Hal59**] it is shown that $[G : G_\alpha] - 1$ of the Schreier generators for G_α are equal to the identity. For instance, if for some point $p \in \alpha^G$ we have that (p, p^s) is an edge of the Schreier tree for α^G, then the Schreier generator $t(p)st(p^s)^{-1}$ is the identity. Thus, the number of non-trivial Schreier generators can be as large as $(|S| - 1)[G : G_\alpha] + 1$, though some of them may be equal to each other.

In our case we calculate the Schreier generators for each G^i, using S^{i-1} in place of S, and since S^{i-1} are precisely the calculated Schreier generators for G^{i-1}, the number of non-trivial Schreier generators for G^i may be as large as

$$1 + (|S| - 1) \prod_{j=0}^{i} \left| \alpha_{j+1}^{G^j} \right| \tag{5.5}$$

Since the orbit sizes are only bounded by $|X|$, we see that (5.5) is exponential in $|X|$, and therefore this method may not be efficient.

5.2. Reducing the number of generators. It is possible to reduce the number of generators at each step, so that our generating set never grows too large. This can be done using Algorithm 3.5, which is due to Sims, and which can also be found in [**Sim03**] and [**CSS99**].

This algorithm reduces the generating set to size $\binom{|X|}{2} \in \mathrm{O}(|X|^2)$. With this algorithm, we can solve our problem using Algorithm 3.6 and Algorithm 3.7. This, however, is not the Schreier-Sims algorithm, which is a more clever and efficient method of performing the same things.

Algorithm 3.5: `BoilSchreierGenerators`

Data: A group G acting on a finite set X, a partial base $B = (\alpha_1, \ldots, \alpha_k)$ and corresponding partial strong generating set S for G, an integer $1 \leq m \leq k$.
Result: A smaller partial strong generating set for G

```
1  begin
2      for i := 1 to m do
3          T := S^{i-1}
4          foreach g ∈ T do
5              foreach h ∈ T do
6                  if α_i^g = α_i^h ≠ α_i then
7                      S := (S \ {h}) ∪ {gh^{-1}}
8                  end
9              end
10         end
11     end
12     return S
13 end
```

Algorithm 3.6: `ComputeBSGS`

Data: A group $G = \langle S \rangle$ acting on a finite set X.
Result: A base and strong generating set for G

```
1  . begin
2      (base, sgs) := GetPartialBSGS(S, ∅)
3      for i := 1 to |base| do
4          (base, sgs) := Schreier(base, sgs, i)
5          sgs := BoilSchreierGenerators(base, sgs, i)
6      end
7      return (base, sgs)
8  end
```

6. Membership testing

We now assume that we know a (complete) base $B = (\alpha_1, \ldots, \alpha_n)$ and a (complete) strong generating set S for our group G, and we present an efficient algorithm for determining if, given an arbitrary group element g, it is true that $g \in G$. The implicit assumption is of course that $G \leq H$ for some large group H and that $g \in H$, but since we are interested in matrix groups this is always true with H being some general linear group.

This algorithm is used in the Schreier-Sims algorithm together with Theorem 3.6, as we shall see later.

Recall that if we have a base then there is an associated stabiliser chain, and as described in section 2, if $g \in G$ we can factorise g as a product of coset representatives $g = u_n u_{n-1} \cdots u_1$ where u_i is the representative of $G^i g$ in G^{i-1}. Moreover,

Algorithm 3.7: `Schreier`

Data: A group G acting on a finite set X, a partial base $B = (\alpha_1, \ldots, \alpha_k)$ and corresponding partial strong generating set S for G, an integer $1 \leq i \leq k$ such that $G^j = H^j$ for $j = 0, \ldots, i-1$.
Result: Possibly extended partial base $B = (\alpha_1, \ldots, \alpha_m)$ and corresponding partial strong generating S set for G such that $G^j = H^j$ for $j = 0, \ldots, i$.
/*Assumes the existence of a function `NewBasePoint`(g) that returns a point $p \in X$ such that $p^g \neq p$ */

```
1  begin
2      T := S^{i-1}
3      tree := ComputeSchreierTree(T, α_i)
4      foreach p ∈ α_i^{H^{i-1}} do
5          foreach s ∈ T do
6              gen := GetSchreierGenerator(tree, p, s)
7              if gen ≠ 1 then
8                  S := S ∪ {gen, gen^{-1}}
9                  if B^{gen} = B then
10                     point := NewBasePoint(gen)
11                     B := B ∪ {point}
12                 end
13             end
14         end
15     end
16     return (B, S)
17 end
```

if we for each $i = 1, \ldots, n$ have computed a Schreier tree T_i for $\alpha_i^{G^{i-1}}$ then we can use Algorithm 3.2 to compute each coset representative.

More specifically, if $g \in G$ then $g = g_1 t(\alpha_1^g)$ where, as before, $t(p)$ is the output of Algorithm 3.2 on the point p and $g_1 \in G^1$. On the other hand, if $g \notin G$ then either $\alpha_1^g \notin \alpha^G$ or $g_1 \notin G_1$. To test if $g \in G$ we can therefore proceed inductively, and first check if $\alpha_1^g \in \alpha^G$ and if that is true then test whether $g_1 \in G_1$. This is formalised in Algorithm 3.8.

The terminology *residue* and *level* is introduced here, with obvious meanings. As can be seen, the algorithm returns the level at which it fails, as this is needed in the Schreier-Sims algorithm. Note that even if all n levels are passed, it might happen that the residue $r \neq 1$ at line 10, which also indicates that $g \notin G$.

In the literature, Algorithm 3.8 is usually referred to as *sifting* or *stripping* of the group element g.

7. The main algorithm

Finally, we can now present the Schreier-Sims algorithm itself. It uses a more efficient method of reducing the number of Schreier generators considered, by making use of Algorithm 3.8.

Algorithm 3.8: `Membership`

Data: A group G acting on a finite set X, a base $B = (\alpha_1, \ldots, \alpha_n)$, a Schreier tree T_i for the orbit $\alpha_{i+1}^{G^i}$ for each $i = 0, \ldots, n-1$, and a group element g.
Result: A residue r and drop-out level $1 \leq l \leq n+1$.

```
1  begin
2      r := g
3      for i := 1 to n do
4          if α_i^r ∉ T_{i-1} then
5              return (r, i)
6          end
7          element := OrbitElement(T_{i-1}, α_i^r)
8          r := r · element^{-1}
9      end
10     return (r, n + 1)
11 end
```

Using the notation from Theorem 3.6, we want to show that $H^i_{\alpha_{i+1}} = H^{i+1}$ for each i, or equivalently that all Schreier generators for H^i are in H^{i+1}. If we proceed from $i = n-1, \ldots, 0$ instead of the other way, then for $H^n = 1$ we obviously already have a base and strong generating set, so we can use Algorithm 3.8 to check if the Schreier generators for $H^{n-1}_{\alpha_n}$ are in H^n.

When we have checked all Schreier generators for $H^{n-1}_{\alpha_n}$ we then have a base and strong generating set for H^{n-1} by Theorem 3.6, since $H^{n-1}_{\alpha_n} = H^n$, and we can therefore proceed inductively downwards. This is shown in Algorithm 3.9 and Algorithm 3.10.

Algorithm 3.9: `ComputeBSGS`

Data: A group $G = \langle S \rangle$ acting on a finite set X.
Result: A base and strong generating set for G

```
1  . begin
2      (base, sgs) := GetPartialBSGS(S, ∅)
3      for i := |base| to 1 do
4          (base, sgs) := SchreierSims(base, sgs, i)
5      end
6      return (base, sgs)
7  end
```

7.1. Matrix groups. As can be seen in the given algorithms, the main part of the Schreier-Sims algorithm is indepedent of the particular type of group, but we are interested in the situation where $G \leq \mathrm{GL}(d, q)$ for some $d \geq 1$ and some $q = p^r$ where p is a prime number and $r \geq 1$. Here, d is called the *degree* of G and is the number of rows (columns) of the matrices in G, and q is the finite field size, ie G contains matrices over $\mathrm{GF}(q) = \mathbb{F}_q$. In this setting, G acts faithfully from the right

Algorithm 3.10: `SchreierSims`

Data: A group G acting on a finite set X, a partial base $B = (\alpha_1, \ldots, \alpha_k)$ and corresponding partial strong generating set S for G, an integer $1 \leq i \leq k$ such that $H^{j-1}_{\alpha_j} = H^j$ for $j = i+1, \ldots, k$, and Schreier trees T^{j-1} for $\alpha_j^{H^{j-1}}$ for $j = i+1, \ldots, k$.

Result: Possibly extended partial base $B = (\alpha_1, \ldots, \alpha_m)$ and corresponding partial strong generating S set for G such that $H^{j-1}_{\alpha_j} = H^j$ for $j = i, \ldots, m$ and Schreier trees T^{j-1} for $\alpha_j^{H^{j-1}}$ for $j = i, \ldots, m$.

/*Assumes the existence of a function `NewBasePoint`(g) that returns a point $p \in X$ such that $p^g \neq p$ */

```
1  begin
2      gens := S^i
3      T^{i-1} := ComputeSchreierTree(gens, α_i)
4      foreach p ∈ α_i^{H^{i-1}} do
5          foreach s ∈ gens do
6              gen := GetSchreierGenerator(T^{i-1}, p, s)
7              if gen ≠ 1 then
8                  (residue, dropout) := Membership(T^i, ..., T^k, gen)
9                  if residue ≠ 1 then
10                     S := S ∪ {gen, gen^{-1}}
11                     if dropout = k + 1 then
12                         point := NewBasePoint(gen)
13                         B := B ∪ {point}
14                     end
15                     for j := r to i + 1 do
16                         (B, S) := SchreierSims(B, S, j)
17                     end
18                 end
19             end
20         end
21     end
22     return (B, S)
23 end
```

on the vector space $X = \mathbb{F}_q^d$, by multiplication of a row vector with a matrix. We denote the standard base in $\mathbb{F}_q^d$ by $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_d$.

What remains to specify is the algorithm `NewBasePoint`, which depends on the particular point set X and the action that is used. To construct this algorithm, we observe that given a matrix $M \neq I \in G$, if $M_{ij} \neq 0$ for some $i \neq j$ then $\mathbf{e}_i M \neq \mathbf{e}_i$ so the row vector $\mathbf{e}_i$ is a point moved by M. If M instead is diagonal, but not a scalar matrix, then $M_{ii} \neq M_{jj}$ for some $i \neq j$, and $(\mathbf{e}_i + \mathbf{e}_j)M \neq \mathbf{e}_i + \mathbf{e}_j$ so this row vector is moved by M. Finally if M is scalar, then $M_{11} \neq 1$ since $M \neq I$ and thus $\mathbf{e}_1 M \neq \mathbf{e}_1$. This translates directly into Algorithm 3.11.

Algorithm 3.11: `NewBasePoint`

Data: A matrix $M \neq I \in G \leq \mathrm{GL}(d, q)$.
Result: A row vector $\mathbf{v} \in \mathbb{F}_q^d$ such that $\mathbf{v}M \neq \mathbf{v}$

```
1  . begin
2      for i := 1 to d do
3          for j := 1 to d do
4              if i ≠ j and M_ij ≠ 0 then
5                  return e_i
6              end
7          end
8      end
9      for i := 1 to d do
10         for j := 1 to d do
11             if i ≠ j and M_ii ≠ M_jj then
12                 return e_i + e_j
13             end
14         end
15     end
16     return e_1
17 end
```

8. Complexity analysis

We now analyse the time complexity of some of the given algorithms, but we will not be interested in space complexity. For the analysis we assume that the external functions `Tree`, `AddChild` and `EdgeLabel`, that depend on the particular datastructure used, take $\Theta(1)$ time. This is a reasonable assumption and it is satisfied in the code for the project, which uses hash tables to implement the Schreier trees. We also assume that the datastructure used for sets is a sorted list, so that elements can be found, added and removed in logarithmic time using binary search. This assumption is satisfied in GAP.

Moreover, since we are working in a matrix group $G = \langle S \rangle \leq \mathrm{GL}(d, q)$ that acts on the vector space $X = \mathbb{F}_q^d$, we know that the multiplication of two group elements takes $\Theta(d^3)$ time and that the action of a group element on a point takes $\Theta(d^2)$ time, under the assumption that the multiplication of two elements from $\mathbb{F}_q$ takes $\Theta(1)$ time. This assumption is reasonable since we are mostly interested in the case when q is not too large, so that one field element can be stored in a machine word and manipulated in constant time. Then we also see that testing equality between two group elements takes $\mathrm{O}(d^2)$ time and testing equality between two points takes $\mathrm{O}(d)$ time.

Consider first Algorithm 3.1. We know that it is a breadth-first search and we know that a simple breadth-first search on a graph $\mathcal{G} = (V, E)$ takes $\mathrm{O}(|V| + |E|)$ time. In our case we have $|E| = |V|\,|S| = |X|\,|S|$, so the edges are dominating. We also see that line 8 takes $\Theta(d^2)$ time and line 11 takes $\mathrm{O}(\log |S|)$ time since the size of the set `children` is bounded by $|S|$. The former line is therefore dominating, and thus we have that Algorithm 3.1 takes $\mathrm{O}(|X|\,|S|\,d^2)$ time.

For Algorithm 3.2 we just note that in a tree with vertex set V, the depth of any node is bounded by $|V|$. In our case the vertices of the Schreier tree is the points of the orbit, which may be the whole of X if the action is transitive. Therefore we see that Algorithm 3.2 takes time $\mathrm{O}(|X|\, d^4)$.

It is obvious that Algorithm 3.11 takes time $\mathrm{O}(d^3)$, and it is also obvious that Algorithm 3.4 takes time $\mathrm{O}(|X|\, d^2 + d^3) = \mathrm{O}(|X|\, d^2)$. In Algorithm 3.3 we see that line 7 takes time $\mathrm{O}(1)$ because the base is just a list and can therefore be augmented in constant time. We see that line 5 takes time $\mathrm{O}(|B|\, d^3)$ since we might have to multiply every base point with the group element and check equality, in the case where it actually fixes the base. Thus, Algorithm 3.3 takes time $\mathrm{O}(|S| \log |S|\, |B|\, d^3)$.

Since Algorithm 3.7 is not used in the GAP package of the project, we skip the complexity analysis of it and its related algorithms. We also skip any detailed complexity of the Schreier-Sims algorithm itself, since we will see that for matrix groups, this is futile anyway. The case of permutation groups is analysed in [**But91**] and the result is that the Schreier-Sims algorithm has time complexity $\mathrm{O}(|X|^5)$.

The essential problem when using the algorithm for matrix groups is then that $X = \mathbb{F}_q^d$ so $|X| = q^d$ and the Schreier-Sims algorithm is thus exponential in d. This is quite disheartening, since it is the case when d grows large that we are interested in, more often than when q is large. However, it is possible to make the implementation fast enough for many practical purposes, so all is not lost.

CHAPTER 4

The probabilistic Schreier-Sims algorithm

It is well-known in computer science that probabilistic algorithms often turn out to be much simpler in structure than the corresponding deterministic algorithms for the same problem, while still producing good solutions. That the algoritms are simpler usually means that they can be made more efficient and are easier to implement.

On the other hand, the drawback with the probabilistic approach is that there is a non-zero probability that the algorithms in some sense can "lie", ie return incorrect solutions. An introduction to the subject and the various complexity classes can be found in [**Pap94**].

The probabilistic Schreier-Sims algorithm is a good example of this situation. It was first described in [**Leo80**], and the idea comes from the following

THEOREM 4.1. *Let G be a group acting on the finite set X, and let $B = (\alpha_1, \ldots, \alpha_k)$ be a partial base and S a partial strong generating set for G. For $i = 1, \ldots, n$, let $G^i = G_{\alpha_1,\ldots,\alpha_i}$, $S^i = S \cap G^i$, $H^i = \langle S^i \rangle$, let $G = G^0 = H^0$ and let T^i be a right transversal for the cosets of $H^{i-1}_{\alpha_i}$ in H^{i-1}. Then $\left|\prod_{i=1}^n T^i\right|$ divides $|G|$.*

PROOF. Since $\left|T^i\right| = [H^{i-1} : H^{i-1}_{\alpha_i}]$ we have

$$
\begin{aligned}
|G| &= \prod_{i=1}^{n} [H^{i-1} : H^i] = \prod_{i=1}^{n} [H^{i-1} : H^{i-1}_{\alpha_i}][H^{i-1}_{\alpha_i} : H^i] = \\
&= \left(\prod_{i=1}^{n} \left|T^i\right|\right) \left(\prod_{i=1}^{n} [H^{i-1}_{\alpha_i} : H^i]\right)
\end{aligned} \tag{0.1}
$$

and thus the theorem follows. □

COROLLARY 4.2. *Let G be a group acting on the finite set X, and let $B = (\alpha_1, \ldots, \alpha_k)$ be a partial base and S a partial strong generating set for G. If B and S are not complete and $g \in G$ is a uniformly random element, then the probability that Algorithm 3.8 returns residue $r \neq 1$ when given g is at least $1/2$.*

PROOF. We see that even if B and S are not complete, we can compute Schreier trees for the orbits $\alpha_{i+1}^{H^i}$ for $i = 0, \ldots, n-1$ and feed them to Algorithm 3.8 which then tries to express g in terms of the corresponding right transversals T^i for $[H^{i-1} : H^{i-1}_{\alpha_i}]$.

Thus, Algorithm 3.8 checks if $g \in \prod_{i=1}^n T^i$ and by Theorem 4.1, if $\prod_{i=1}^n T^i$ is not the whole G, then it contains at most half of the elements of G. Therefore, since g is uniformly random, we have $\Pr[g \notin \prod_{i=1}^n T^i] \geq 1/2$. □

This suggests a probabilistic algorithm for computing a base and a strong generating set. Given a partial base and partial strong generating set for G, compute

Schreier trees and chose random elements uniformly from G. Use Algorithm 3.8 on each element, and if it returns a non-trivial residue, add it to the partial SGS, and possibly augment the base.

If the base and SGS are complete, then of course the residue will be trivial. On the other hand, if the base and SGS are not complete, then by Corollary 4.2, the probability that k consecutive random elements have trivial residues is less than 2^{-k}. We can thus choose a large enough k for our purposes and assume that the base and SGS are complete when we have stripped k consecutive random elements to the identity. This is formalised in Algorithm 4.1.

Algorithm 4.1: `RandomSchreierSims`

Data: A group G acting on a finite set X, a partial base $B = (\alpha_1, \ldots, \alpha_k)$ and corresponding partial strong generating set S for G, an integer $m \geq 1$ and Schreier trees T^{j-1} for $\alpha_i^{H^{i-1}}$ for $i = 1, \ldots, k$.

Result: Possibly extended partial base $B = (\alpha_1, \ldots, \alpha_n)$ and corresponding partial strong generating S set for G such that m consecutive random elements have been stripped to identity with respect to B and S.

/*Assumes the existence of a function `NewBasePoint`(g) that returns a point $p \in X$ such that $p^g \neq p$. */

/*Assumes the existence of a function `Random`(G) that returns a uniformly random element from the group G. */

```
1  begin
2      sifts := 0
3      while sifts < m do
4          element := Random(G)
5          (residue, dropout) := Membership(T^1, ..., T^k, element)
6          if residue ≠ 1 then
7              S := S ∪ {element, element^-1}
8              if dropout = k + 1 then
9                  point := NewBasePoint(element)
10                 B := B ∪ {point}
11                 k := k + 1
12             end
13             for i := 1 to k do
14                 T^i := ComputeSchreierTree(S^i, α_i)
15             end
16             sifts := 0
17         else
18             sifts := sifts + 1
19         end
20     end
21     return (B, S)
22 end
```

Indeed, Algorithm 4.1 is simpler than Algorithm 3.10 and implementations are usually faster. However, it is not clear how the algorithm `Random` should be constructed. First of all, we need random bits, and the generation of pseudo-random bits is an old problem in computer science, see [**Pap94**] and [**Knu97**], which we will not dwell into. Instead, we will assume that uniformly random bits are available.

1. Random group elements

The generation of random group elements is also difficult in general, and an introduction to the topic can be found in [**Bab97**]. If one knows a base and strong generating set one can easily generate random elements by using the factorisation described in section 2, select random elements from each transversal and multiply them. In our case we do not yet have a base and an SGS, since that is what we are trying to compute, and then the only known algorithm that can *provably* generate (almost) uniformly random group elements is described in [**Bab91**] and runs in time $\mathrm{O}(d^{10}(\log q)^5)$ for subgroups of $\mathrm{GL}(d, q)$, and this is too slow for our purposes.

The practical algorithms are instead *heuristics*, see [**ACG+99**], for which there are no *proven* guarantees for any uniformly randomness, but instead experimentation and statistical analysis has shown them to be good. It appears that the most successful algorithm is the *product replacement* algorithm, also called "Shake", which is originally an idea by Charles Leedham-Green and Leonard Soicher, and is described in [**CLGM+95**].

We will not dwell deeper into the analysis of this algorithm, but the algorithm itself is quite simple. Given a group $G = \langle g_1, \ldots, g_n \rangle$, the "Shake" algorithm maintains a global variable $S = (a_1, \ldots, a_m) \in G^m$ for some $m \geq n$, and then each call to Algorithm 4.2 returns a proposed random group element.

Algorithm 4.2: `Shake`

Data: The global state $S = (a_1, \ldots, a_m) \in G^m$
Result: An element in G that hopefully is a good approximation of a uniformly random element from G.
/*Assumes the existence of a function `RandomInteger`(k) that returns a uniformly random integer from the set $\{0, \ldots, k\}$ */

```
1  begin
2      i := RandomInteger(m)
3      repeat
4          j := RandomInteger(m)
5      until i ≠ j
6      if RandomInteger(1) = 0 then
7          b := a_i a_j
8      else
9          b := a_j a_i
10     end
11     a_i := b
12     return b
13 end
```

Evidently, the algorithm is very cheap in terms of time and space. The questions that remain are how the state S should be initialised and how m should be chosen, and the ability of the algorithm to generate uniformly random group elements are highly dependent on the answers. In [**CLGM$^+$95**] the suggestion is that $m = \max(10, 2n + 1)$ and that S is initialised to contain the generators $g_1, \ldots, g_n$, the rest being identity elements. Moreover, the state should be initiliased by calling Algorithm 4.2 a number K of times and discarding the results, and the suggestion is that $K \geq 60$.

In [**NP01**] this algorithm is also described, as well as a variant of it, called "Rattle", which is due to Leedham-Green. The algorithms are compared and "Rattle" is found to be slightly better, but its running time is a bit higher. In GAP, there is an implementation of the "Shake" algorithm, which is used in this project. An implementation of "Rattle" was also made, but it turned out that it was not efficient enough.

CHAPTER 5

The Schreier-Todd-Coxeter-Sims algorithm

Coset enumeration is one of the oldest algorithms in group theory, and it was first described in [**TC36**]. Given a finitely presented group G and $H \leq G$, the aim of coset enumeration is to construct the list of the cosets of H in G, ie to find a set X with a point $p \in X$ such that G acts transitively on X and such that $H = G_p$. The set X is usually taken to be $\{1, \ldots, n\}$ for some n and $p = 1$. If the index of H is not finite, then the algorithm will usually not terminate.

The coset enumeration is a trial-and-error process, and quite a few strategies have been developed over the years. We will not be concerned about the actual algorithm, since it is beyond the scope of this report, and there is a good implementation in GAP, which is used in this project. A more detailed description of coset enumeration can be found in [**Sim03**].

However, it is important to be aware of the overall structure of coset enumeration. It works by letting the elements of G act on X and introducing new cosets when necessary. Then it may happen that some cosets turn out to be the same and thus are identified in the list. The structure of the algorithm makes it possible to exit prematurely when a given number of cosets have been defined, and this can be used if one knows an upper bound on the number of cosets.

In our case, we want to use the last case of Theorem 3.6. At level i of Schreier-Sims algorithm, we know that the base and SGS are complete for higher levels, and want to verify that $H^{i+1} = H^i_{\alpha_{i+1}}$, which by the theorem is equivalent to $[H^i : H^{i+1}] = \left|\alpha_{i+1}^{H^i}\right|$ and since we can compute a Schreier tree for $\alpha_{i+1}^{H^i}$ we know the orbit size. Thus, if we can compute $[H^i : H^{i+1}]$ using coset enumeration and it turns out to be equal the orbit size, it is unnecessary to compute any Schreier generators and check for membership.

All this is formalised in Algorithm 5.1 where we use the option to exit the coset enumeration after $M\left|\alpha_{i+1}^{H^i}\right|$ cosets have been defined, for some rational number $M \geq 1$, since this certainly is an upper bound. In our implementation, the number M can be specified by the user, and the default is $M = 6/5$. This value comes from [**Leo80**], where the algorithm was introduced and some experimentation was carried out to determine a good value of M.

The Schreier-Todd-Coxeter-Sims algorithm is known to perform particularly good when the input partial base and SGS are actually already complete. Thus, it can be used for *verification* of the output from a probabilistic algorithm, and in this project it is mainly used in that way, to verify the output from the probabilistic algorithm described earlier.

Algorithm 5.1: `SchreierToddCoxeterSims`

Data: A group G acting on a finite set X, a partial base $B = (\alpha_1, \ldots, \alpha_k)$ and corresponding partial strong generating set S for G, an integer $1 \leq i \leq k$ such that $H^{j-1}_{\alpha_j} = H^j$ for $j = i+1, \ldots, k$, and Schreier trees T^{j-1} for $\alpha_j^{H^{j-1}}$ for $j = i+1, \ldots, k$.

Result: Possibly extended partial base $B = (\alpha_1, \ldots, \alpha_m)$ and corresponding partial strong generating S set for G such that $H^{j-1}_{\alpha_j} = H^j$ for $j = i, \ldots, m$ and Schreier trees T^{j-1} for $\alpha_j^{H^{j-1}}$ for $j = i, \ldots, m$.

/*Assumes the existence of a function `NewBasePoint`(g) that returns a point $p \in X$ such that $p^g \neq p$. */

/*Assumes the existence of a function `ToddCoxeter`(U_1, U_2, k) that performs coset enumeration on $G = \langle U_1 \rangle$ and $H = \langle U_2 \rangle \leq G$, exiting when k cosets have been defined. */

```
1  begin
2      gens := S^i
3      T^{i-1} := ComputeSchreierTree(gens, α_i)
4      foreach p ∈ α_i^{H^{i-1}} do
5          foreach s ∈ gens do
6              table := ToddCoxeter(S^i, S^{i+1}, |T^i| + 1)
7              if |table| = |T^i| then
8                  return (B, S)
9              end
10             gen := GetSchreierGenerator(T^{i-1}, p, s)
11             if gen ≠ 1 then
12                 (residue, dropout) := Membership(T^i, ..., T^k, gen)
13                 if residue ≠ 1 then
14                     S := S ∪ {gen, gen^{-1}}
15                     if dropout = k + 1 then
16                         point := NewBasePoint(gen)
17                         B := B ∪ {point}
18                     end
19                     for j := r to i + 1 do
20                         (B, S) := SchreierToddCoxeterSims(B, S, j)
21                     end
22                 end
23             end
24         end
25     end
26     return (B, S)
27 end
```

CHAPTER 6

Implementation and optimisation

We will now consider some more practical issues regarding the actual implementation and optimisation of the code in the project.

In GAP, a very fast implementation of the Schreier-Sims algorithm already exists. It is described in [**Ser03**] and is a heuristic based on the algorithm described in [**BCFÁS91**]. As with many other algorithms in GAP, it works only with permutation groups, so for a general group G it first calculates a faithful permutation representation, ie an injective homomorphism $\lambda : G \to \mathrm{Sym}(X)$, for some finite set X, which exists due to the well-known theorem by Cayley. It then work with the image $\lambda(G)$, or rather with $H \leq S_{|X|}$ such that $H \cong \lambda(G)$.

Maybe the most important motivation for the GAP package developed in this project is the idea that if G is a matrix group, and one work with the matrices directly instead of first converting to permutations, then perhaps one could use the additional structure, that is otherwise thrown away, and come up with a faster algorithm. The code is therefore created with the basic assumption that all group elements are invertible matrices over a finite field, and it uses the internal GAP representation of such matrices.

1. Schreier tree creation

An important issue is how the Schreier trees are managed. Actually, what we really want is the transversals defined by the Schreier trees, so a possibility is to store not a tree, but just a list of all points and for each point the element that moves the root to this point. This can be realised as a tree with height 1, where the edge labels are not the generators, as in the case of a Schreier tree, but the group elements, so in general this will take up more memory than the Schreier tree. On the other hand, to find a coset representative in a Schreier tree we need to follow a path from a point to the root, which takes logarithmic time on average, and may take linear time if the tree is not properly balanced, but in the case where we store the coset representatives as edge labels, then of course it takes constant time to find them later.

To conclude, we have a trade-off between time and space, and in this project, both of the above strategies are available to the user of the package. The concrete representation of the trees are as hash tables, where the keys are the points and the values are the edge labels of the unique edges directed towards the root, ie we use "back-pointers". This makes it possible to perform in constant time the common task of checking if a given point is in the orbit defined by a Schreier tree, since this is just a hash table lookup.

1.1. Extending vs creating. If we consider Algorithm 3.10 in more detail, then we notice that the sets S^i are only augmented if they are changed. Therefore,

at line 3 we could save the tree that is computed and, at the next time we arrive there, only extend the tree using the new generators, if there are any.

This may or may not give a faster algorithm. Evidently, there will be less work in Algorithm 3.1 to extend a Schreier tree than to create a new one, but it may happen that the tree becomes more balanced if it is recomputed using all generators than if it is extended. If the tree is not balanced, then Algorithm 3.2 will take more time, and this is in fact the function where most time is spent. In [**Ser03**] it is claimed that this problem with unbalanced trees are more common for matrix groups, and empirical studies in this project has shown that in most cases, it is better to recompute the Schreier trees every time.

1.2. Shallow trees. There are algorithms for the creation of Schreier trees that are guaranteed to make the Schreier trees *shallow*, ie balanced, so that the they have worst-case logarithmic height. This is crucial if one wants to have good worst-case complexity, and two algorithms are described in [**Ser03**], one deterministic and one probabilistic. In this project the deterministic algorithm have been implemented, which is also described in [**BCFÁS91**]. It is too complicated to be included here in more detail, but the essential idea is to choose a different set of edge labels, rather than the given generators.

2. Orbit sizes

As have been noted earlier, using the Schreier-Sims algorithm for a matrix group $G \leq \mathrm{GL}(d, q)$ is in a sense doomed from the beginning, since the complexity is exponential in d when G acts on $\mathbb{F}_q^d$. One manifestation of doom in this case is that the orbits may become huge, something that does not happen for permutation groups. This will make our Schreier trees huge and a large amount of Schreier generators must be created, so the algorithm will be slow.

2.1. Alternating actions. To avoid large orbits, one can use another action of G. However, only if the action is faithful is it a permutation representation of G, and this is needed if the Schreier-Sims algoritm is going to work, otherwise we will calculate a base and strong generating set for the quotient of G with the kernel of the action.

In [**But76**], a clever trick was introduced where base points are chosen alternatingly as one-dimensional subspaces (lines) and vectors from those lines. When a vector $v = (v_1, \ldots, v_d) \in \mathbb{F}_q^d$ is chosen as base point, it is preceded in the base by the line $\langle v \rangle$. The action of G on the lines is known as the *projective action* and of course it is not faithful, but since the next point v is from the kernel this does not matter. Also, the subspace $\langle v \rangle$ has a canonical representative $(1, v_2 v_1^{-1}, \ldots, v_d v_1^{-1})$, which is trivial to find from v. Therefore, in terms of time, the projective action is not particularly more expensive than the action on points.

Now, if $k = \left|v^G\right|$ then $\left|\langle v \rangle^G\right| \left|v^{G_{\langle v \rangle}}\right| = k$ and $G_{\langle v \rangle}$ is the stabiliser of the line containing v. This implies that if $u \in v^{G_{\langle v \rangle}}$ then u is also on that line, so $u = mv$ where $m \in \mathbb{F}_q$. We see that $M = \left\{m \in \mathbb{F}_q \mid mv \in v^{G_{\langle v \rangle}}\right\}$ is a subgroup of $\mathbb{F}_q^*$ and thus $l = \left|v^{G_{\langle v \rangle}}\right| = |M|$ divides $q - 1$.

Instead of one orbit of size k we therefore have two orbits of size k/l and l, and since $k \geq k/l + l$ whenever $l \geq 2$ we will for example need to compute and sift fewer Schreier generators. On the other hand, our base will probably be longer

when using this trick, and empirical studies in this project has shown that it is not always a good idea, but it nevertheless implemented and can be used.

2.2. Eigenspaces. Another method for producing smaller orbits is described in [**MO95**]. The idea is to choose the base points to be eigenvectors of the given generator matrices.

Recall from linear algebra that the *characteristic polynomial* of a matrix $A \in \mathrm{GL}(d, q)$ is $c_A(x) = \det(xI - A)$ where I is the identity matrix, and an *eigenvalue* of A is a root of $c_A(x)$. Normally, and *eigenvector* is defined as a vector $v \in \mathbb{F}_q^d$ such that $v^A = \lambda v$ where λ is an eigenvalue of A, or equivalently $v^{g(A)} = 0$ where $g(x)$ is a linear factor of $c_A(x)$. Here we use a more general version of the latter definition, where we allow factors of higher degree as well as linear factors of $c_A(x)$.

We see that for a linear factor $g(x) = x - \lambda$ and a corresponding eigenvector v, the size of the orbit $v^{\langle A \rangle}$ is a divisor of $q-1$, since if $u \in v^{\langle A \rangle}$ then $u = v^{\alpha A} = \alpha\lambda v$, and as before the possible values of α form a subgroup of $\mathbb{F}_q^*$. More generally, for a factor of $c_A(x)$ of degree m, the size of the orbit $v^{\langle A \rangle}$ for an eigenvector v is bounded above by $q^m - 1$. To get the smallest possible orbits, we should therefore choose eigenvectors corresponding to factors of as small degree as possible, and linear factors are easy to find since they correspond to eigenvalues, which are easy to compute.

Note that we have only considered the orbits of groups generated by single matrices, and the orbit of a matrix group $G = \langle S \rangle$ need not be small just because we choose as base point an eigenvector of one of the generators in S, but if we choose a base point that is an eigenvector of several generators, then it turns out that the orbit size are more often small. In [**MO95**] these issues are investigated and experimented with in some detail, and a heuristic for finding base points that will hopefully give small orbits is developed. This has also been implemented in GAP, and in this project it is possible to use that algorithm. However, it is not always a good idea to use it, since there is some overhead, and it is not certain that the orbit sizes will actually be smaller.

3. Further developments

The package contains implementations of two more advanced algorithms, the so-called *Verify* routine by Sims, which is an algorithm for verifying if a proposed base and SGS are complete, and the nearly linear time algorithm for finding a base and an SGS. The complete descriptions of these algorithms are beyond the scope of this report, but for completeness we include brief accounts on them.

3.1. The Verify routine. This algorithm is due to Sims, and has never been published, but it is described in [**Ser03**]. Given a group $G = \langle S \rangle$ acting on the finite set X, a point $\alpha \in X$ and a subgroup $H = \langle S' \rangle \leq G_\alpha$, it checks whether $H = G_\alpha$. If this is not the case, the algorithm computes $g \in G_\alpha \setminus H$ to witness that. To check a whole proposed base and SGS, one then uses the third case of Theorem 3.6 and checks each level.

The algorithm is quite involved, both theoretically and when it comes to implementing it. It has a recursive nature, inducting on $S \setminus S'$, and it involves things like changing base points and computing block systems.

3.2. The nearly linear time algorithm. For permutation groups G acting on X, there is an algorithm for computing a base and an SGS that runs in nearly linear time, ie linear in $|X|$ except for some logarithmic factor of $|X|$ and $|G|$. It is the best known algorithm in terms of time complexity, and is described in [**BCFÁS91**] as well as in [**Ser03**] and it is probabilistic. The complexity is achieved by using shallow Schreier trees, a fast probabilistic verification algorithm, and by sifting only a few randomly selected Schreier generators. Otherwise, the algorithm is quite similar to the algorithms we have described earlier for computing a base and an SGS.

The random Schreier generators are computed using *random subproducts* which are described in [**Ser03**] as well as by selecting random group elements, and the algorithm relies on some theorems relating the number of Schreier generators to compute to the given probability of correctness.

CHAPTER 7

Performance

As observed earlier, one of the objectives of the project was to make an implementation that hopefully would be faster than the one already existing in GAP. To determine if this objective was met, the implementation has been benchmarked and compared with the built-in GAP implementation.

The algorithm has been used to compute a base and SGS for some matrix groups that are easy to construct in GAP, and the generating sets that were used were the standard generating sets from the GAP library. The main test groups were classical groups: the general and special linear groups $\mathrm{GL}(d, q)$ and $\mathrm{SL}(d, q)$ and the general and special orthogonal groups $\mathrm{GO}(d, q)$ and $\mathrm{SO}(d, q)$, for various (small) d and q. The algorithm was also tested on some Suzuki groups $\mathrm{Sz}(q)$ (where q is a non-square power of 2) and some Ree groups $\mathrm{Ree}(q)$ (where $q = 3^{1+2m}$ for some $m > 0$).

Another, perhaps more realistic, test of the performance was done by running the algorithm on randomly formed sets of invertible matrices of a given dimension over a given finite field.

The benchmarks was carried out on two quite standard PC computers, the first test on a computer with an AMD Athlon CPU running at 2 GHz and with 1 GB of physical RAM, and the second test on a computer with an Intel Pentium 4 CPU running at 2.8 GHz and also with 1 GB of physical RAM. It is likely that these were the only important parameters, since GAP is mainly CPU intensive, and swapping was avoided duringd the benchmark, so the hard disk speed should be a negligible factor. The GAP installation tests reported GAP4stones values of 194624 and 253581, respectively.

The details of the benchmarks is shown in the appendix. During the first test, the existing algorithm was the faster one for most groups, but for some groups our implementation was faster, most notably for the Suzuki groups. It should be noted that all running times for our implementation comes from using the same algorithm options, and it is possible to get better times, especially for the smaller groups, by elaborating with other options. The second test also indicated that the existing implementation is faster in most cases.

In terms of memory, no rigorous benchmark has been performed, but during the above benchmark some simple checks of the amount of memory allocated by the GAP process were performed. It seemed that our implementation consumed less memory overall, and in some cases the difference was large. Indeed, since we wanted to avoid swapping during the benchmark, no larger Suzuki group than $\mathrm{Sz}(32)$ could be checked during the first test, since the existing implementation ran out of memory. Our implementation was far from having such problems, consuming no more than about 50 MB for the Suzuki groups.

The conclusion must therefore be that the project is a small success.

APPENDIX A

Benchmark

Here are details of the results of the first benchmark. The first column shows the test groups, as they are written in GAP, and the other columns show the execution time in milliseconds. The method used to compare the algorithms was to compute the order of the input group using the orbit sizes in the computed base and SGS, so to measure the existing implementation we used a command like

```
gap> Size(Group(GeneratorsOfGroup(GL(4, 4))));
gap> benchmark_time := time;
```

TABLE 1. Benchmark results

Group	Project	GAP
$\mathrm{SL}(2,2)$	110	170
$\mathrm{GO}(+1,2,2)$	20	10
$\mathrm{GO}(-1,2,2)$	20	0
$\mathrm{GL}(2,4)$	80	20
$\mathrm{SL}(2,4)$	60	10
$\mathrm{GO}(+1,2,4)$	40	0
$\mathrm{GO}(-1,2,4)$	40	10
$\mathrm{GL}(2,3)$	80	30
$\mathrm{SL}(2,3)$	50	0
$\mathrm{GO}(+1,2,3)$	50	10
$\mathrm{GO}(-1,2,3)$	50	0
$\mathrm{SO}(+1,2,3)$	20	0
$\mathrm{SO}(-1,2,3)$	30	10
$\mathrm{GL}(2,5)$	70	10
$\mathrm{SL}(2,5)$	40	10
$\mathrm{GO}(+1,2,5)$	50	10
$\mathrm{GO}(-1,2,5)$	50	0
$\mathrm{SO}(+1,2,5)$	50	10
$\mathrm{SO}(-1,2,5)$	40	0
$\mathrm{SL}(3,2)$	50	10
$\mathrm{GO}(0,3,2)$	40	0
$\mathrm{GL}(3,4)$	170	20
$\mathrm{SL}(3,4)$	110	20
$\mathrm{GO}(0,3,4)$	90	10
$\mathrm{GL}(3,3)$	70	10
$\mathrm{SL}(3,3)$	100	0
$\mathrm{GO}(0,3,3)$	60	10
$\mathrm{SO}(0,3,3)$	50	0

TABLE 2. Benchmark results

Group	Project	GAP
$GL(3,5)$	190	60
$SL(3,5)$	110	40
$GO(0,3,5)$	70	10
$SO(0,3,5)$	70	20
$SL(4,2)$	130	10
$GO(+1,4,2)$	40	10
$GO(-1,4,2)$	60	0
$GL(4,4)$	1480	30
$SL(4,4)$	690	80
$GO(+1,4,4)$	90	30
$GO(-1,4,4)$	170	20
$GL(4,3)$	210	30
$SL(4,3)$	410	20
$GO(+1,4,3)$	80	20
$GO(-1,4,3)$	80	10
$SO(+1,4,3)$	80	0
$SO(-1,4,3)$	80	10
$GL(4,5)$	1600	50
$SL(4,5)$	790	190
$GO(+1,4,5)$	100	60
$GO(-1,4,5)$	140	50
$SO(+1,4,5)$	140	60
$SO(-1,4,5)$	100	60
$SL(5,2)$	560	10
$GO(0,5,2)$	90	10
$GL(5,4)$	26440	220
$SL(5,4)$	4650	440
$GO(0,5,4)$	230	100
$GL(5,3)$	890	30
$SL(5,3)$	5090	130
$GO(0,5,3)$	100	40
$SO(0,5,3)$	150	40
$GL(5,5)$	24500	410
$SL(5,5)$	12380	1450

TABLE 3. Benchmark results

Group	Project	GAP
$GO(0,5,5)$	470	370
$SO(0,5,5)$	390	380
$SL(6,2)$	1250	40
$GO(+1,6,2)$	250	20
$GO(-1,6,2)$	350	10
$GL(6,4)$	205430	460
$SL(6,4)$	89580	4030
$GO(+1,6,4)$	3880 580	
$GO(-1,6,4)$	3700	550
$GL(6,3)$	3850	100
$SL(6,3)$	16890	660
$GO(+1,6,3)$	340	90
$GO(-1,6,3)$	340	200
$SO(+1,6,3)$	520	100
$SO(-1,6,3)$	480	100
$GL(6,5)$	292220	84030
$SL(6,5)$	227830	82010
$GO(+1,6,5)$	5410	2050
$GO(-1,6,5)$	1710	1440
$SO(+1,6,5)$	3190	1460
$SO(-1,6,5)$	3520	1390
$Sz(8)$	180	1170
$Sz(32)$	4390	133860
$Sz(128)$	1084510	> 3600000
$Ree(27)$	816590	687780

Here are the results of the second benchmark. The first column shows the finite field size, the second column the matrix dimension and the third column is the size of the random generating set that was formed. Each generating set was computed using the given number of calls to `RandomInvertibleMat`. For each line in the table, 20 generating sets with the given parameters were formed, the algorithm was executed on these sets, and the average time over these 20 executions is shown in the table in the last two columns.

TABLE 4. Benchmark results

Field	Dim	Set	Project	GAP
2	2	1	25	16
2	2	2	23	4
2	2	3	23	2
2	2	4	23	2
2	2	5	23	2
2	2	6	23	2
2	2	7	23	3
2	2	8	23	2
2	2	9	24	3
2	2	10	23	3
2	3	1	34	2
2	3	2	42	3
2	3	3	49	3
2	3	4	52	3
2	3	5	53	4
2	3	6	58	4
2	3	7	60	4
2	3	8	68	4
2	3	9	62	5
2	3	10	68	4
2	4	1	46	3
2	4	2	87	4
2	4	3	117	5
2	4	4	120	6
2	4	5	126	6
2	4	6	124	7
2	4	7	145	7
2	4	8	151	7
2	4	9	161	8
2	4	10	168	8

TABLE 5. Benchmark results

Field	Dim	Set	Project	GAP
2	5	1	58	3
2	5	2	208	10
2	5	3	231	11
2	5	4	301	12
2	5	5	294	13
2	5	6	357	14
2	5	7	384	15
2	5	8	355	15
2	5	9	407	16
2	5	10	471	17
2	6	1	75	4
2	6	2	612	24
2	6	3	530	27
2	6	4	704	28
2	6	5	1033	32
2	6	6	903	34
2	6	7	1209	37
2	6	8	1055	41
2	6	9	1300	44
2	6	10	1158	45
2	7	1	86	8
2	7	2	1229	24
2	7	3	1675	20
2	7	4	2289	20
2	7	5	2183	25
2	7	6	2404	23
2	7	7	2556	24
2	7	8	2477	25
2	7	9	2963	29
2	7	10	3343	27
4	2	1	37	3
4	2	2	55	4
4	2	3	60	5
4	2	4	61	5
4	2	5	62	5
4	2	6	64	6
4	2	7	69	7
4	2	8	70	7
4	2	9	73	7
4	2	10	75	8
4	3	1	59	5
4	3	2	118	12
4	3	3	133	14
4	3	4	152	17
4	3	5	158	19
4	3	6	175	21
4	3	7	185	22
4	3	8	215	25
4	3	9	216	27
4	3	10	226	28

TABLE 6. Benchmark results

Field	Dim	Set	Project	GAP
4	4	1	75	15
4	4	2	430	26
4	4	3	576	28
4	4	4	589	29
4	4	5	749	28
4	4	6	714	31
4	4	7	838	33
4	4	8	912	38
4	4	9	1019	36
4	4	10	1085	38
4	5	1	109	52
4	5	2	3141	170
4	5	3	4248	119
4	5	4	3660	106
4	5	5	5126	116
4	5	6	3895	124
4	5	7	4416	134
4	5	8	5154	137
4	5	9	5002	147
4	5	10	6285	156
4	6	1	193	212
4	6	2	27943	655
4	6	3	33664	455
4	6	4	30882	478
4	6	5	40467	506
4	6	6	36702	534
4	6	7	35179	570
4	6	8	41211	593
4	6	9	47733	630
4	6	10	46698	661
4	7	1	8786	19238
4	7	2	335656	95870
4	7	3	337232	100999
4	7	4	371067	98525
4	7	5	381631	94616
4	7	6	463195	94353
4	7	7	463824	102045
4	7	8	460079	104262
4	7	9	485832	115177
4	7	10	497838	99240
3	2	1	39	3
3	2	2	47	3
3	2	3	49	3
3	2	4	52	3
3	2	5	51	4
3	2	6	53	4
3	2	7	46	4
3	2	8	54	4
3	2	9	54	5
3	2	10	47	5

TABLE 7. Benchmark results

Field	Dim	Set	Project	GAP
3	3	1	59	4
3	3	2	92	7
3	3	3	98	8
3	3	4	101	9
3	3	5	111	9
3	3	6	114	11
3	3	7	126	12
3	3	8	126	13
3	3	9	136	12
3	3	10	143	15
3	4	1	74	7
3	4	2	244	22
3	4	3	278	26
3	4	4	310	29
3	4	5	366	32
3	4	6	334	35
3	4	7	415	39
3	4	8	452	40
3	4	9	475	45
3	4	10	454	47
3	5	1	82	15
3	5	2	1029	44
3	5	3	1110	28
3	5	4	1692	29
3	5	5	1361	36
3	5	6	1612	34
3	5	7	1752	35
3	5	8	1887	40
3	5	9	1928	39
3	5	10	2197	42
3	6	1	111	44
3	6	2	6517	105
3	6	3	5772	131
3	6	4	7463	117
3	6	5	6184	104
3	6	6	7111	128
3	6	7	7191	117
3	6	8	8478	120
3	6	9	11749	127
3	6	10	10142	134
3	7	1	385	128
3	7	2	33440	591
3	7	3	36465	586
3	7	4	35244	506
3	7	5	35492	331
3	7	6	43913	351
3	7	7	47698	364
3	7	8	45104	382
3	7	9	46772	401
3	7	10	62226	421

TABLE 8. Benchmark results

Field	Dim	Set	Project	GAP
5	2	1	46	3
5	2	2	59	5
5	2	3	61	6
5	2	4	64	6
5	2	5	64	8
5	2	6	65	9
5	2	7	68	9
5	2	8	73	10
5	2	9	72	10
5	2	10	77	11
5	3	1	59	8
5	3	2	147	14
5	3	3	157	15
5	3	4	178	15
5	3	5	182	14
5	3	6	203	15
5	3	7	226	17
5	3	8	240	17
5	3	9	258	19
5	3	10	269	20
5	4	1	78	35
5	4	2	957	94
5	4	3	1123	73
5	4	4	1038	65
5	4	5	1300	68
5	4	6	1337	72
5	4	7	1563	78
5	4	8	1703	81
5	4	9	1815	93
5	4	10	1864	91
5	5	1	166	158
5	5	2	10330	414
5	5	3	11236	508
5	5	4	11779	402
5	5	5	13499	355
5	5	6	15476	375
5	5	7	14578	475
5	5	8	14204	428
5	5	9	14994	450
5	5	10	15487	472
5	6	1	903	3568
5	6	2	180644	86753
5	6	3	195962	91533
5	6	4	210412	84445
5	6	5	216448	84051
5	6	6	225522	80495
5	6	7	212674	76792
5	6	8	258668	81668
5	6	9	269393	97520
5	6	10	287699	87331

Bibliography

[ACG+99] Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Alberto Marchetti-Spaccamela, Marco Protasi, and Viggo Kann, *Complexity and Approximation : Combinatorial Optimization Problems and Their Approximability Properties*, ch. 10, pp. 321–351, Springer, Berlin Heidelberg, 1999.

[Bab91] László Babai, *Local expansion of vertex-transitive graphs and random generation in groups*, Proc. 23rd ACM Symp. Theory of Computing (Los Angeles), Association for Computing Machinery, 1991, pp. 164–174.

[Bab97] ———, *Randomization in group algorithms: Conceptual questions*, Groups and Computation II, American Mathematical Society DIMACS Series, vol. 28, 1997, pp. 1–17.

[BB96] John A. Beachy and William D. Blair, *Abstract algebra*, 2nd ed., Waveland Press, Illinois, 1996.

[BCFÁS91] László Babai, Gene Cooperman, Larry Finkelstein, and Ákos Seress, *Nearly linear time algorithm for permutation groups with a small base*, Proc. of International Symposium on Symbolic and Algebraic Computation, ISSAC '91, ACM Press, New York, 1991, pp. 200–209.

[BH99] Martin R. Bridson and André Haefliger, *Metric spaces of non-positive curvature*, ch. 1, pp. 6–8, Springer, 1999.

[Big89] Norman L. Biggs, *Discrete mathematics*, Oxford University Press, 1989.

[But76] Gregory Butler, *The Schreier algorithm for matrix groups*, Proc. of ACM Symposium on symbolic and algebraic computation, SYMSAC '76, Association for Computing Machinery, New York, 1976, pp. 167–170.

[But91] ———, *Fundamental algorithms for permutation groups*, Lecture Notes in Computer Science, vol. 559, Springer, 1991.

[CH92] John Cannon and George Havas, *Algorithms for groups*, Australian Computer Journal **24** (1992), 59–68.

[CLGM+95] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and Eamonn A. O'Brien, *Generating random elements of a finite group*, Commutative Algebra (1995), no. 23, 4931–4948.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to algorithms*, The MIT Electrical Engineering and Computer Science Series, The MIT Press, 1990.

[CSS99] Hans Cuypers, Leonard H. Soicher, and Hans Sterk, *Working with finite groups*, Some Tapas of Computer Algebra (Arjeh M. Cohen, Hans Cuypers, and Hans Sterk, eds.), Algorithms and Computation in Mathematics, vol. 4, Springer, 1999, pp. 184–207.

[GAP] *Groups, Algorithms and Programming*, http://www.gap-system.org/.

[Hal59] Marshall Hall, *The theory of groups*, Macmillan, New York, 1959.

[Jen97] Robert J. Jenkins, *Hash functions for hash table lookup*, Dr. Dobb's Journal (1997), http://burtleburtle.net/bob/hash/evahash.html.

[Knu97] Donald E. Knuth, *The Art of Computer Programming : Seminumerical algorithms*, 3rd ed., vol. 2, ch. 3, Addison-Wesley, Reading, Massachusetts, 1997.

[Leo80] Jeffrey S. Leon, *On an algorithm for finding a base and strong generating set for a group given by generating permutations*, Mathematics of Computation **35** (1980), no. 151, 941–974.

[MO95] Scott H. Murray and Eamonn A. O'Brien, *Selecting base points for the Schreier-Sims algorithm for matrix groups*, Journal of Symbolic Computation **19** (1995), no. 6, 577–584.

[Mur93] Scott H. Murray, *The Schreier-Sims algorithm*, Master's thesis, Australian National University, 1993, http://www.win.tue.nl/~smurray/research/.

[Neu95] Joachim Neubüser, *An invitation to computational group theory*, Groups '93 (Galway/St, Andrews), London Mathematical Society Lecture Note Series, vol. 212, Cambridge University Press, 1995, pp. 457–475.

[NP01] Alice C. Niemeyer and Cheryl E. Praeger, *Complexity and computation in matrix groups*, Aspects of complexity (Rod Downey and Denis Hirschfeldt, eds.), de Gruyter Series in Logic and its Applications, vol. 4, de Gruyter, Berlin, 2001, pp. 87–113.

[Pap94] Christos H. Papadimitriou, *Computational complexity*, 1 ed., Addison-Wesley, January 1994.

[Ros94] John S. Rose, *A course on group theory*, Dover, 1994.

[Sch27] Otto Schreier, *Die Untergruppen der freien Gruppen*, Abh. Math. Sem. Univ. Hamburg **5** (1927), 161–183.

[Ser97] Ákos Seress, *An introduction to computational group theory*, Notices of the American Mathematical Society **44** (1997), 671–679, http://www.math.ohio-state.edu/~akos/.

[Ser03] ———, *Permutation group algorithms*, Cambridge Tracts in Mathematics, vol. 152, Cambridge University Press, 2003.

[Sim70] Charles C. Sims, *Computational methods in the study of permutation groups*, Computatioanl problems in Abstract Algebra, Pergamon Press, Oxford, 170, pp. 169–183.

[Sim03] ———, *Computational group theory*, Computer Algebra Handbook (Johannes Grabmeier, Erich Kaltofen, and Volker Weispfenning, eds.), Springer, http://www.math.rutgers.edu/~sims/publications/, 2003.

[TC36] J.A. Todd and H. S. M. Coxeter, *A practical method for enumerating cosets of a finite abstract group*, Proceedings of the Edinburgh Mathematical Society, vol. 5, 1936, pp. 26–34.