Otto-von-Guericke University Magdeburg
Faculty of Computer Science
Department for Simulation and Graphics

Diploma Thesis

Fundamental Permutation Group Algorithms for Symmetry Computation

Author:

Thomas Rehn

January 12, 2010

Supervisors:

Prof. Dr. rer. nat. habil. Stefan Schirra

Otto-von-Guericke University Magdeburg
Faculty of Computer Science
Postfach 4120, D–39016 Magdeburg
Germany

Priv.-Doz. Dr. rer. nat. habil. Achill Schürmann

Institute of Applied Mathematics
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Rehn, Thomas:
Fundamental Permutation Group Algorithms for Symmetry Computation
Diploma Thesis, Otto-von-Guericke University Magdeburg, January 2010.

Contents

List of Figures

List of Algorithms

1 Introduction

1.1 Motivation

Over the last 40 years powerful algorithmic methods have been developed for many problems in algebra. Along with others, the so called computational group theory that deals with algorithms for problems of group theory has grown to a mature tool (cf. [CH92]). With more and more practical algorithms getting available and ever-growing computer power, the usage of group theory has reached into many areas of mathematics and other natural sciences. The book [Ker99], for instance, provides a very good overview of the applicability of group theory to combinatorial objects from many applications.

From highly successful single-purpose applications like graph theory (cf. [McK81] and the software [`nauty`]) more general methods have evolved for the computation with symmetries or auto- and isomorphisms. These are applicable to matrices, codes, designs, graphs and groups alike [Leo84, BL85, Leo91, McK98, Bri00]. Other areas have profited from this progress and are able to handle or exploit symmetries in their problems, which helps to extend the realm of practically solvable problems and instances. Just to name a few of these applications, [Jun03] examines symmetry of petri-nets. [KÖ06] describe methods to classify codes and designs isomorphism-free. [Mar09] shows in his survey the impact of symmetry to integer programming. [BSS09] enumerate vertices and rays of polyhedra up to the immanent symmetry of the geometric object. Mathematically, this symmetry computation part translates into computations with and search in permutation groups.

All these examples have in common that they use different tools to handle the basic symmetry part of their real computational task. Either they implement permutation group algorithms in own code for this fundamental recurring problem or use well-tested group theoretic software like the open-source algebra system GAP [`GAP`] or the commercial Magma [`Magma`]. These software packages, however, lack proper and simple APIs to be used from C or C++, still the language of choice in many projects, and cause huge dependencies. There also exists an open source C implementation for such permutation group problems by Leon from 1991 [Leo91], but it is designed as a pure stand-alone program and not as callable library. Moreover, it has too many memory leaks and errors to be used as such. Since 2008 there also have been efforts of the open source Sage project [`Sage`] to provide implementations of automorphism and permutation group algorithms as part of their package (cf. [Mila, Milb]). To date only the former part is complete.

There are many excellent books available that cover group algorithms, for example [But91], [Ser03] and [HEO05], but these rather aim at more sophisticated fields of computational group theory and do not go into implementation details, which may be of interest for practitioners from other areas. This is especially true for the allegedly fast partition backtrack methods of Leon, which most books cover, if at all, only in a very abstract way, hiding the difficulties an implementation may face. This thesis presents the very basic techniques to solve the group theoretic part of common problems in symmetry computations along with an implementation of all algorithms as a C++ library, christened PermLib.

We begin in Chapter 2 with a look at fundamental data structures and algorithms to work with permutation groups. Because permutation groups usually consist of a huge number of elements they are not given as a complete set of permutations, but only a few generating elements are known, from which all other elements can be derived. This already causes problems with the simple task of group membership testing. We will look at a data structure called base and strong generating set which allows a representation of a group such that this and other very basic problems can be solved efficiently.

Having learned about suitable structures to store permutation groups, in Chapter 3 we will deal with searching for specific subgroups or elements in a permutation group. Therefore we will analyze two different general purpose backtracking algorithms. As an example we will study the special cases of set stabilizers and group intersections.

In Chapter 4 we will look at specific implementation details and the results of experiments with the PermLib. There we will see the algorithms' and implementation's limits, potential and potential pitfalls, some of which have to the best of the author's knowledge not been extensively discussed in public. Chapter 5 provides a summary of our findings as well as an outlook into possible future work.

1.2 Permutations and notation

A fundamental object that we will work with all the time are permutations. Permutations are bijections from a set Ω to itself. This set could be vertices or facets of a polyhedron, vertices of a graph, variables in an equation system and many other things, but we identify Ω without loss of generality with the set of numbers $\{1, \dots, n\}$, $n := |\Omega|$. There are two different ways to specify the bijection of a permutation. The first way is to give the image of each $\omega \in \Omega$ explicitly, the so called image form. In this thesis we will use the second way, that is cycle notation: a cycle is a sequence $\omega_1, \omega_2, \dots, \omega_k$ such that

- ω_2 is the image of ω_1,
- ω_3 is the image of ω_2,
- $\dots$
- ω_k is the image of ω_{k-1} and
- ω_1 is the image of ω_k.

We can write a permutation as a sequence of disjoint cycles, which is called cycle form.

Example 1.1. Consider $\Omega = \{1, 2, 3, 4, 5, 6\}$ and the permutation g with the following image:

$$\begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 3 & 1 & 4 & 6 & 5 \end{array}$$

where the first row lists all elements of Ω and the second row their images under g. We observe that g has three disjoint cycles $(1\,2\,3)$, (4) and $(5\,6)$. We omit the cycle of length 1 and write $g = (1\,2\,3)(5\,6)$, or equivalently $g = (5\,6)(1\,2\,3)$, because for disjoint cycles order does not matter.

As we use cycle notation for permutations, we write $()$ for the identity permutation. Permutations on the same set with the concatenation operation build a group. Because the notation of this concatenation resembles a multiplicative operation we usually say that permutations are multiplied. The interested reader may find in [But91, Ch. 2] more examples of permutations and also a very good introduction to groups, which is not given here. In the following we will establish notation for groups that we use throughout this thesis.

If a group G is finitely generated by some elements $g_1, \dots, g_k$ we write $G = \langle g_1, \dots, g_k \rangle$. For two groups G, H we denote with $G \leq H$ that G is a subgroup of H. The number of elements in G is called **order** of G and we denote it by $|G|$. Furthermore, we stick to Knuth [Knu91] for his compact notation of group inverses, so g^- shall be the inverse element of g.

As for this thesis we are especially interested in symmetries within a finite set of objects, we always consider a permutation group $G \leq \mathrm{Sym}(\Omega)$ acting on a finite set Ω, where G is a subgroup of the group $\mathrm{Sym}(\Omega)$ of all permutations of Ω. We call the minimal size $|\Omega|$ such that G is properly defined the **degree** of G. Of course we can always trivially extend $G \leq \mathrm{Sym}(\Omega)$ to a subgroup of some $\mathrm{Sym}(\Omega')$ where $\Omega' \supset \Omega$. Lowercase Greek letters will denote elements of Ω and we use lower- and uppercase Latin letters for elements and subgroups of the group $\mathrm{Sym}(\Omega)$.

We write α^g for the action of $g \in G$ on $\alpha \in \Omega$ and this action will be left-associative, i.e. $\alpha^{gh} = (\alpha^g)^h$. The **orbit** of α under G is the set of images $\alpha^G := \{\alpha^g \,:\, g \in G\}$.

Example 1.2. Consider $\Omega = \{1, \dots, 6\}$ and $G = \langle a, b \rangle$ generated by $a = (1\,4\,5)(2\,3\,6)$, $b = (2\,3\,1\,6)$, denoted in cycle form. Then we have $b^- = (2\,6\,1\,3)$, $1^a = 4$, $1^{ba} = 6^a = 2$ and orbits $1^{\langle b \rangle} = \{1, 2, 3, 6\}$ and $1^G = \Omega$.

Let $H, K \leq G$ be subgroups of G and $g \in G$. Then we define the **right coset** $Hg := \{hg \,:\, h \in H\}$ and analogously the left coset $gH := \{gh \,:\, h \in H\}$. The **double coset** HgK is given by $HgK := \{hgk \,:\, h \in H, k \in K\}$. As becomes immediately clear from the definition, two cosets Hg_1, Hg_2 are either the same or disjoint. Because every $g \in G$ is in one coset, G is partitioned by its cosets. Thus it makes sense to define a right (left) **transversal** $U \subseteq G$ for G modulo H as a set containing exactly one representative of every H-right (left) coset of G, including the identity $() \in U$.

For every subgroup H we can define the **index** $|G : H|$ of H in G as the number of right (or equivalently: left) cosets of H in G. With this notation at hand we can remind ourselves of Lagrange's Theorem.

Theorem 1.3 (Lagrange's Theorem). Let G be a finite group and H a subgroup of G. Then $|G| = |G : H| \cdot |H|$.

Proof. As we have already seen G can be partitioned into its cosets modulo H. Let U be a transversal for G modulo H. Then we have

$$|G| = \sum_{u \in U} |Hu|. \tag{1.1}$$

Furthermore, for every coset Ha and $g \in G$ we have $|Ha| = |Hag|$. So it holds especially that $|Ha| = |Ha(a^- b)| = |Hb|$. It follows that every summand in (1.1) is of the same size, resulting in $|G| = |U| \cdot |H()| = |G : H| \cdot |H|$. □

An immediate consequence of this proof is the following corollary.

Corollary 1.4. Let G and H as above and U a transversal for G modulo H. Every $g \in G$ can uniquely be written as $g = uh$ where $u \in U$ and $h \in H$.

Example 1.5. Let $\Omega = \{1, 2, 3, 4\}$ and $G = S_4 = \langle (1\,2), (2\,3), (3\,4) \rangle$ the symmetric group of four elements. Then $H := \langle (1\,2), (2\,3) \rangle \leq G$ is a subgroup of G. There are $|G : H| = \frac{|G|}{|H|} = \frac{24}{6} = 4$ right cosets of H in G: the right cosets $H()$, $H(3\,4)$, $H(2\,4)$ and $H(1\,4)$, corresponding to the four possible images of the remaining point 4. Thus the four elements $U := \{(), (3\,4), (2\,4), (1\,4)\}$ form a transversal of G modulo H.

The **stabilizer** G_α of α in G is defined as the set $G_\alpha := \{g \in G : \alpha^g = \alpha\}$ and forms a subgroup of G. In the same manner we can define the pointwise stabilizer of a tuple $(\alpha_1, \ldots, \alpha_k)$ as $G_{(\alpha_1,\ldots,\alpha_k)} := \{g \in G : \forall 1 \leq i \leq k : \alpha_i^g = \alpha_i\}$ and the stabilizer of a set $\{\alpha_1, \ldots, \alpha_k\}$ as $G_{\{\alpha_1,\ldots,\alpha_k\}} := \{g \in G : \forall 1 \leq i \leq k : \exists j : \alpha_i^g = \alpha_j\}$. Note the difference in notation between pointwise stabilizer $G_{(\alpha_1,\ldots,\alpha_k)}$ and setwise stabilizer $G_{\{\alpha_1,\ldots,\alpha_k\}}$.

Example 1.6. Let again $\Omega = \{1, 2, 3, 4\}$ and $G = S_4 = \langle (1\,2), (2\,3), (3\,4) \rangle$. Then $G_1 = \langle (2\,3), (3\,4) \rangle$, $G_{(1,2)} = \langle (3\,4) \rangle$ and $G_{\{1,2\}} = \langle (1\,2), (3\,4) \rangle$.

2 Introduction to Bases and Strong Generating Sets

To work with permutation groups we need a suitable structure to represent them on the computer. Usually we are given a group $G \leq \mathrm{Sym}(\Omega)$ by a list of generators $S = \{s_1, s_2, \ldots, s_k\}$ with $G = \langle S \rangle$. If we want to search for group elements with specific properties as in Chapter 3 we are confronted with a problem: We know that each element is a product of elements of S but we have no efficient means to systematically enumerate all elements of G, for example for an exhaustive search. The reverse problem also exists: given some $x \in \mathrm{Sym}(\Omega)$, we cannot decide whether $x \in G$. This is a real obstacle for systematic search approaches.

In 1970 Sims introduced the concept of a base for computations with permutation groups to overcome these difficulties (cf. [Sim70, Sim71a]). Similarly to a base in vector spaces group elements have a unique representation relative to it. Such a base also allows easy group membership testing, in this context usually called "sifting".

First we will look at the fundamental concepts of bases. Before we see how to set up a base and the related strong generating sets for a permutation group we have to examine different ways to store orbits and transversals and to solve the group membership problem using bases. After the base construction with the Schreier-Sims algorithm we will look at two other tools for the efficient use of bases: an improved algorithm to compute transversals and base change algorithms. Again similarly to vector spaces, one certain base is more comfortable in some contexts to work with than another, so we will see how we can transform a base without re-constructing it from scratch.

2.1 Definitions

Definition 2.1. Let G be a finite permutation group acting on the set $\{1, \ldots, n\}$. We call a sequence of elements $B := (\beta_1, \beta_2, \ldots, \beta_m)$ a **base** for G if the only element of G to fix B pointwise is the identity.

For a base B we denote by $G^{[i]} := G_{(\beta_1, \ldots, \beta_{i-1})}$ the pointwise stabilizer of the $i-1$ first base elements $(\beta_1, \ldots, \beta_{i-1})$ which form a subgroup chain, the **stabilizer chain**:

$$G = G^{[1]} \geq G^{[2]} \geq \cdots \geq G^{[m]} \geq G^{[m+1]} = \langle () \rangle. \tag{2.1}$$

If every $G^{[i+1]}$ is a proper subgroup of $G^{[i]}$ we call the base B **nonredundant**.

The cosets of $G^{[i]}$ modulo $G^{[i+1]}$ are closely related to the orbits $\beta_i^{G^{[i]}}$. For two cosets $G^{[i+1]}a = G^{[i+1]}b$ with $a, b \in G^{[i]}$ we have $a = hb$ for $h \in G^{[i+1]}$ and thus $\beta_i^a = \beta_i^{hb} = \beta_i^b$ because h stabilizes β_i. Also the reverse direction holds: From $\beta_i^a = \beta_i^b$ we can immediately conclude that the two cosets $G^{[i+1]}a$, $G^{[i+1]}b$ are the same. So we can build a transversal for $G^{[i]}$ modulo $G^{[i+1]}$, which contains one representative for every coset, by looking at elements generating the orbit $\Delta^{(i)} := \beta_i^{G^{[i]}}$. For every $\beta \in \Delta^{(i)}$ let $u_\beta \in G^{[i]}$ be an element that maps β_i to β, i. e. $\beta_i^{u_\beta} = \beta$. Then it follows from our considerations that $U^{(i)} := \{u_\beta \,:\, \beta \in \Delta^{(i)}\}$ is a (right) transversal for $G^{[i]}$ modulo $G^{[i+1]}$. We call $\Delta^{(i)}$ the **i-th fundamental orbit**.

Example 2.2. Let $\Omega = \{1, 2, 3, 4\}$ and $G = S_4$ the symmetric group of four elements. The sequence $(4, 3)$ is not a base because the permutation $(1\,2)$ stabilizes the tuple pointwise. If we add 2 to the list we get a base $(4, 3, 2)$ because the image of $3 = 4 - 1$ points already determines a permutation: $(4, 3, 2)^g = (4, 3, 2)$ implies $g = ()$.

The stabilizer chain consists of

$$\begin{aligned}
G^{[1]} &= G = S_4 \\
G^{[2]} &= G_{(4)} = \langle (1\,2), (2\,3) \rangle \cong S_3 \\
G^{[3]} &= G_{(4,3)} = \langle (1\,2) \rangle \cong S_2 \\
G^{[4]} &= G_{(4,3,2)} = \langle () \rangle
\end{aligned}$$

and we see that the base $(4, 3, 2)$ is nonredundant.

For $\Delta^{(2)} = \beta_2^{G^{[2]}}$ we obtain $\Delta^{(2)} = 3^{S_3} = \{1, 2, 3\}$. We can build a transversal $U^{(2)}$ from the elements $u_1 = (3\,1)$, $u_2 = (3\,2)$ and $u_3 = ()$. Note that for constructing a transversal we have some degrees of freedom. We could, for instance, also choose $u_1 = (1\,2\,3)$ because still $3^{u_1} = 1$. In Section 2.2.1 we will look at algorithms for orbit and transversal construction in detail.

Back in our general setting, we can repeatedly apply Corollary 1.4 of Lagrange's Theorem on page 4 to the stabilizer chain (2.1) and obtain that every $g \in G$ can uniquely be decomposed into

$$g = u_m u_{m-1} \cdots u_2 u_1, \quad \text{for some } u_i \in U^{(i)}, \tag{2.2}$$

where $U^{(i)}$ are transversals as introduced above. This especially means that we can read the group order from the transversal sizes

$$|G| = \prod_{i=1}^{m} |G^{[i]} : G^{[i+1]}| = \prod_{i=1}^{m} |U^{(i)}|. \tag{2.3}$$

For a nonredundant base we can thus also bound the base size by $2^{|B|} \leq |G| \leq n^{|B|}$ because $1 < |G^{[i]} : G^{[i+1]}| = |U^{(i)}| = |\Delta^{(i)}| \leq n$. With log denoting the logarithm to base 2, this is equivalent to

$$\frac{\log |G|}{\log n} \leq |B| \leq \log |G|.$$

So far we do not know how to compute $G^{[i]}$ and the related orbits and transversals $\Delta^{(i)}$ and $U^{(i)}$. An important concept to facilitate this is a strong generating set.

Definition 2.3. Let S be a generating set for a finite permutation group G with base B. The set S is a **strong generating set (SGS)** for G relative to B if it contains generators for all $G^{[i]}$, that is

$$G^{[i]} = \langle S \cap G^{[i]} \rangle, \quad \text{for } 1 \leq i \leq m + 1. \tag{2.4}$$

For brevity, we call a pair B, S of a strong generating set S relative to a base B a **BSGS**.

Example 2.4. Let $\Omega = \{1, 2, 3, 4\}$ and $G = S_4$. The set $\{(1\,2), (2\,3), (3\,4)\}$ obviously is a strong generating set relative to the base $B := (4, 3, 2)$ because it contains generators for all subgroups of the stabilizer chain (cf. Example 2.2). The set $T := \{(1\,2\,3\,4), (3\,4)\}$ also generates G, but it is not a strong generating set relative to B: For every $t \in T$ it holds that $\beta_1^t = 4^t \neq 4 = \beta_1$, so $T \cap G^{[2]}$ is empty. Hence $\langle T \cap G^{[2]} \rangle = \langle () \rangle \neq G^{[2]} = \langle (1\,2), (2\,3) \rangle$ violates the defining equality of a strong generating set (2.4).

Bases may also consist of only a single element. Consider $\Omega = \{1, \dots, n\}$ and the cyclic group $G = C_n := \langle (1\,2\,3\,\dots\,n) \rangle$. Then $B_i := (i)$ is a base for each $i \in \Omega$ because every permutation in C_n except the identity moves every point of Ω. Thus $\{(1\,2\,\dots\,n)\}$ trivially is a strong generating set relative to every base B_i.

Having a BSGS, we know generators for each $G^{[i]}$ of the stabilizer chain. Hence we can efficiently compute the transversals $U^{(i)}$ used in (2.2), which gives us a powerful instrument to deal with permutation groups in practice. In the following section we will see how to actually compute orbits and transversals and see a first important application of a BSGS: group membership testing.

2.2 Elementary algorithms & data structures

2.2.1 Orbits and transversals

Before we start with transversals we examine a straightforward algorithm, Algorithm 2.1, to compute the orbit α^G of a point α under action of G. To see that it works correctly we first observe that we only add β^s to the orbit Δ if β has previously been in Δ. So by induction we have $\Delta \subseteq \alpha^G$. On the other hand, induction on the length t of an element $g = s_1 \cdots s_t$ in terms of generators $s_i \in S$ shows that every α^g is added to Δ, showing also the reverse inclusion $\Delta \supseteq \alpha^G$.

From a complexity point of view, the algorithm runs in $O(|S| \cdot |\alpha^G|)$ time if we can test membership in Δ in $O(1)$ time. If we, as usual, identify Ω with the set $\{1, \dots, n\}$, storing orbit membership in an array or bitset will satisfy this assumption.

```
Input: $G = \langle S \rangle$ permutation group acting on $\Omega$, $\alpha \in \Omega$
Output: $\alpha^G$
$\Delta \leftarrow \{\alpha\}$
for $\beta \in \Delta$, $s \in S$ do
    if $\beta^s \notin \Delta$ then
        $\Delta \leftarrow \Delta \cup \{\beta^s\}$
    end
end
return $\Delta$
```

Algorithm 2.1: Orbit

Often we require not only the orbit α^G but also a transversal of G mod G_α. That means, we would like to have an element u_β for each $\beta \in \alpha^G$ with $\beta = \alpha^{u_\beta}$. For handling these transversals we have two alternatives: computing and storing them explicitly or using a so called Schreier tree.

Definition 2.5. Let $S \subseteq \mathrm{Sym}(\Omega)$ be a generating set for G and $L \subseteq G$ be a suitable label set. A labeled, directed tree with vertex set $V = \alpha^G$, edge set E and edge label set $L \subseteq \mathrm{Sym}(\Omega)$ such that

- for every $\beta \in V$ there is a unique path from β to α.
- for every edge $\overrightarrow{\beta_1\,\beta_2} \in E$ there exists $l \in L$ with $\beta_1^l = \beta_2$ and $\overrightarrow{\beta_1\,\beta_2}$ has label l

is called a **Schreier tree** for α^G.

Example 2.6. Consider $a = (1\,2\,5)$, $b = (1\,4)(3\,5)$ and $G = \langle a, b\rangle$. Figure 2.1 shows a Schreier tree for 1^G with label set $S^- = \langle a^-, b^-\rangle$.

$$\begin{tikzcd}
 & 1 & \\
2 \arrow[ur, "a^-"] & & 4 \arrow[ul, "b^-"'] \\
5 \arrow[u, "a^-"] & & \\
3 \arrow[u, "b^-"] & &
\end{tikzcd}$$

Figure 2.1: A Schreier tree for the orbit of 1 under action of $G = \langle a, b\rangle$ as in Example 2.6

With a Schreier tree we can compute the desired transversal elements u_β by following the path from β to α and multiplying the edge labels along the way. We can modify Algorithm 2.1 to construct a Schreier tree. We just have to keep track of the occurring pairs (β, β^s), that may be used as edges of a Schreier tree. Algorithm 2.2 contains this modification to construct a Schreier tree with label set $L = S^- := \{s^- \in S\}$. Note that we use S^- as label set because we construct the tree based on S starting at the root, but all edges are directed towards the root, the other way round.

A disadvantage of a tree constructed in this way is that it is not balanced and has a worst-case height of n. For example, Schreier trees for orbits of elements under action of a cyclic group will degenerate into lists. In Section 2.4 we will take a look at a balanced alternative with worst-case height $O(\log|G|)$, which can be established by choice of a better label set.

A suitable representation of Schreier trees on the computer are so called **Schreier vectors**. Without loss of generality let $\Omega = \{1, \ldots, n\}$ and $L = \{l_1, \ldots, l_k\}$ be a label set. A Schreier vector is an array of length n that has in its i-th cell either

- j if the pair $\overrightarrow{i\,i^{l_j}}$ is an edge of the Schreier tree, or
- the value 0 if $i = \alpha$, or
- a special marker if $i \notin \alpha^G$.

Figure 2.2 gives a Schreier vector encoding for the tree in Figure 2.1.

By storing the transversal only as pointers to the label set, we need $O(|L| \cdot n + |\alpha^G|)$ memory. Here the second summand stems from storing a pointer to the outgoing edge

```
Input: $G = \langle S \rangle$ permutation group acting on $\Omega$, $\alpha \in \Omega$
Output: $\alpha^G$, labeled edge set $E$ for Schreier tree with label set $S^- := \{s^- \in S\}$
1 $\Delta \leftarrow \{\alpha\}$
2 $E \leftarrow \emptyset$
3 for $\beta \in \Delta, s \in S$ do
4     if $\beta^s \notin \Delta$ then
5         $\Delta \leftarrow \Delta \cup \{\beta^s\}$
6         add edge $\overrightarrow{\beta^s \beta}$ with label $s^-$ to $E$
7     end
8 end
9 return $\Delta, E$
```

Algorithm 2.2: Orbit and Schreier tree

array index	1	2	3	4	5
cell content	0	1	2	2	1

Figure 2.2: Schreier vector encoding for Figure 2.1 with $l_0 = ()$, $l_1 = a^-$, $l_2 = b^-$

label for each orbit element except the root. The first summand including the size of the label set $|L|$ usually does not play a role. It is very common to use $L = S^-$ and it makes sense for implementations to store the often required inverses S^- along with the group generators S anyway. Because the resulting tree has a possible height of n computing a transversal element may take n multiplications of permutations acting on n points, resulting in a worst-case $\Omega(n^2)$ time requirement. Constructing the transversal works in $O(|\alpha^G|)$ time, so constructing orbit and transversal together take $O(|S||\alpha^G|)$ time.

An alternative to Schreier trees is storing the transversal elements explicitly in an array. For every $\beta := \alpha^{s_1 \cdots s_t}$ that we have computed we store in the β-th cell the whole product $s_1 \cdots s_t$, or for performance reasons straight its inverse $s_t^- \cdots s_1^-$. This saves us the edge multiplications when we need a specific transversal element, but costs extra memory because there possibly are a lot of permutations to store. Moreover, multiplications already occur during the construction phase, which makes the setup slower. The explicit variant consumes $O(|\alpha^G| \cdot n)$ memory and is constructed in $O(|\alpha^G||S|n)$ time together with the orbit. The advantage of $O(1)$ transversal access may be well worth the additional memory and slower construction time. For larger n the memory and construction time requirements may become a serious impediment. Figure 2.3 continues Example 2.6 and shows an explicit transversal.

array index	1	2	3	4	5
cell content	()	$a^- = (1\,5\,2)$	$b^-a^-a^- = (1\,4\,2\,5\,3)$	$b^- = (1\,4)(3\,5)$	$a^-a^- = (1\,2\,5)$

Figure 2.3: Explicit transversal as vector for Figure 2.1

2.2.2 Sifting

Given a BSGS for a group, we can efficiently test for membership by a procedure called **sifting**. If and only if $g \in G$ we can find the decomposition (2.2), $g = u_m u_{m-1} \cdots u_1$, into transversal elements $u_i \in U^{(i)}$. To find the decomposition we try to find successively $u_i \in U^{(i)}$, $1 \leq i \leq m$ such that

$$B^g = B^{u_m u_{m-1} \cdots u_1}. \tag{2.5}$$

Here we write B^g for the component-wise acting of $g \in G$ on $(\beta_1, \ldots, \beta_m)$ as a tuple: $B^g = (\beta_1^g, \ldots, \beta_m^g)$. If we succeed in finding u_i such that (2.5) holds and the equality $g = u_m u_{m-1} \cdots u_1$ is true then clearly $g \in G$. On the other hand, if we do not find the u_i or $g \neq u_m u_{m-1} \cdots u_1$ then g cannot be in G because elements of G are uniquely determined by the image of the base points, as proven by the following lemma.

Lemma 2.7. Let $G \leq \mathrm{Sym}(\Omega)$ with a base $B := (\beta_1, \ldots, \beta_m)$. Then for every $g, h \in G$ we have $B^g = B^h$ if and only if $g = h$.

Proof. $B^g = B^h$ is equivalent to $B^{gh^-} = B$. By definition of a base, the only element to fix B pointwise is the identity, so we must have $gh^- = ()$. □

We can find the factors u_i as follows: As $u_2, \ldots, u_m \in G_{\beta_1}$, they fix β_1. Thus the image of β_1 under $u_m u_{m-1} \cdots u_1$ is uniquely determined by u_1, so we look for $u_1 \in U^{(1)}$ such that $\beta_1^g = \beta_1^{u_1}$. After we have fixed u_1 we proceed to the image of β_2, which is determined by $u_2 u_1$. Thus we look for $u_2 \in U^{(2)}$ such that $\beta_2^{gu_1^-} = \beta_2^{u_2}$. We continue in this manner until we have found $u_m \in U^{(m)}$ such that $\beta_m^{gu_1^- u_2^- \cdots u_{m-1}^-} = \beta_m^{u_m}$, or cannot find a suitable $u_j \in U^{(j)}$ before. The product $h := gu_1^- u_2^- \cdots u_j^-$ up to the index $0 \leq j \leq m$ of the last properly found factor u_j is called the **siftee** of g. We say g **sifts through** if the factorization succeeds, i. e. $j = m$ and $h = ()$, and hence $g \in G$. Algorithm 2.3 gives a formal description of this process.

```
Input: $B = (\beta_1, \ldots, \beta_m)$ base, $\Delta$ basic orbits, $U$ transversal system for a
       permutation group acting on $\Omega$, $g \in \mathrm{Sym}(\Omega)$
Output: siftee $h \in \mathrm{Sym}(\Omega)$, sift index $i$
$h \leftarrow g$
for $i = 1$ to $m$ do
    $\beta \leftarrow \beta_i^h$
    if $\beta \notin \Delta^{(i)}$ then
        return $h, i - 1$
    end
    find $u_\beta \in U^{(i)}$ with $\beta_i^{u_\beta} = \beta$
    $h \leftarrow h u_\beta^-$
end
return $h, m$
```

Algorithm 2.3: Sifting

With this fast group membership testing we have almost all ingredients together to look at our first BSGS construction algorithm.

2.3 BSGS construction with Schreier-Sims algorithm

In his paper [Sim70] from 1970 Sims devised a straightforward algorithm to construct a BSGS based on a lemma of Schreier (Lemma 2.8 below). In this section we analyze a variant of this Schreier-Sims algorithm due to [Ser03, Sec. 4.2]. We shall need the following two observations for our analysis. The first lemma presents a way to compute generators of a stabilizer. The second lemma gives us a criterion by which we can verify if a given generator set is a strong generating set.

Lemma 2.8 (Schreier generators). Let $G = \langle X \rangle \leq \mathrm{Sym}(\Omega)$ and $\alpha \in \Omega$. Let U be a transversal for G modulo G_α and $u_\beta \in U$ be the transversal element mapping α to β. Then

$$G_\alpha = \langle \{ u_\beta x u_{\beta^x}^{-} \;:\; \beta \in \alpha^G,\, x \in X \} \rangle \tag{2.6}$$

We call these generators **Schreier generators** for G_α.

Proof. It suffices to show that every element $h \in G_\alpha$ is generated by $T := \{ u_\beta x u_{\beta^x}^{-} :$ $\beta \in \alpha^G,\, x \in X\}$ because $\langle T \rangle \leq G_\alpha$ by definition of T. So let $h = x_1 \cdots x_k \in G_\alpha$ with $x_i \in X$ arbitrary. We now apply a sequence of transformations until h can easily be seen to be of the form $h = t_1 \cdots t_k$, $t_i \in T$. Our intermediate elements h_j will be of the form

$$h_j = t_1 \cdots t_j u_{\gamma_{j+1}} x_{j+1} x_{j+2} \cdots x_k$$

We start with $h_0 := x_1 \cdots x_k = u_\alpha x_1 \cdots x_k$. Given h_j, we set $t_{j+1} := u_{\gamma_{j+1}} x_{j+1} u^{-}_{\gamma_{j+1}^{x_{j+1}}}$ and $\gamma_{j+2} := \gamma_{j+1}^{x_{j+1}}$, ensuring $h_{j+1} = h_j = h$. We can iterate this process until we reach $h_k = t_1 t_2 \cdots t_k u_{\gamma_{k+1}}$. Because $h_k = h \in G_\alpha$ and $(t_1 t_2 \cdots t_k) \in \langle T \rangle \leq G_\alpha$ we must have $u_{\gamma_{k+1}} \in U \cap G_\alpha$. Since U is a transversal modulo G_α the element $u_{\gamma_{k+1}}$ must be the identity. Thus we have h in the desired form $h = t_1 t_2 \cdots t_k$, showing the inclusion $G_\alpha \leq \langle T \rangle$, hence $G_\alpha = \langle T \rangle$. □

Lemma 2.9. Let $B := \{\beta_1, \beta_2, \ldots, \beta_m\} \subseteq \Omega$ and $G \leq \mathrm{Sym}(\Omega)$. For $1 \leq j \leq m+1$ define $G^{[j]} := G_{(\beta_1, \ldots, \beta_{j-1})}$ as before and let $S_j \subseteq G^{[j]}$ with $\langle S_j \rangle \geq \langle S_{j+1} \rangle$. If $\langle S_1 \rangle = G$, $S_{m+1} = \emptyset$, and for $1 \leq j \leq m$

$$\langle S_j \rangle_{\beta_j} = \langle S_{j+1} \rangle \tag{2.7}$$

holds then B is a base for G and $S := \bigcup_{j \leq m} S_j$ is a strong generating set for G relative to B.

Proof. We use induction on $|\Omega|$. For $|\Omega| = 1$ we have only $()$ as group generator and nothing more to proof. So let Ω be of arbitrary size n and be the statement of the lemma true for $|\Omega| \leq n-1$. According to Definition 2.3 of an SGS we have to verify that $G^{[i]} = \langle S \cap G^{[i]} \rangle$ holds for $2 \leq i \leq n$. We first look at the case $i = 2$. We obtain from (2.7), with $j = 1$, and $S_2 \subseteq G^{[2]}$ that

$$G_{\beta_1} = \langle S_2 \rangle = \langle S_2 \cap G^{[2]} \rangle \leq \langle S \cap G_{\beta_1} \rangle \leq G_{\beta_1} \tag{2.8}$$

Because the left-most and right-most terms are equal all inner relations are likewise fulfilled with equality. Hence (2.4) holds for $i = 2$.

For $i > 2$ we can apply the lemma to the case $B' = (\beta_2, \dots, \beta_m)$, $S' = \bigcup_{2 \leq j \leq m} S_j$ and $G' = G_{\beta_1}$ so that we have a group acting on $n - 1$ elements. By doing this, we obtain that $G'_{(\beta_2,\dots,\beta_{i-1})} = \langle S' \cap G'_{(\beta_2,\dots,\beta_{i-1})} \rangle$. This implies

$$G^{[i]} = (G_{\beta_1})_{(\beta_2,\dots,\beta_{i-1})} = \langle S' \cap G_{(\beta_1,\dots,\beta_{i-1})} \rangle \leq \langle S \cap G_{(\beta_1,\dots,\beta_{i-1})} \rangle \leq G_{(\beta_1,\dots,\beta_{i-1})} = G^{[i]} \tag{2.9}$$

So (2.8) and (2.9) show that the SGS condition (2.4) is fulfilled for all i. Thus S is a strong generating set relative to the base B. □

Finally we are ready to construct a base and strong generating set for $G = \langle X \rangle$ with the Schreier-Sims algorithm. We proceed by extending a list $B = (\beta_1, \dots, \beta_k)$ and sets S_i that approximate a generating set for $G^{[i]}$, maintaining $\langle S_i \rangle \geq \langle S_{i+1} \rangle$ for all i. We say our construction is **up to date above level** j when additionally (2.7) holds for all $j < i \leq k$. When we are up to date above level 0 it follows from Lemma 2.9 that we have found a BSGS.

To start the algorithm we choose a $\beta_1 \in \Omega$ which is moved by at least one generator in X. We set $B = (\beta_1)$ and $S_1 := X$ and we are up to date above level 1.

When we are up to date above level j, we test whether (2.7) holds for $i = j$. The inclusion $\langle S_j \rangle_{\beta_j} \geq \langle S_{j+1} \rangle$ is always fulfilled by our construction as we ensure that $S_i \subseteq G^{[i]}$ and hence that β_j is invariant under S_{j+1}. So we need only to test the opposite inclusion $\langle S_j \rangle_{\beta_j} \leq \langle S_{j+1} \rangle$. This can be done by checking that all generators of $\langle S_j \rangle_{\beta_j}$ lie in $\langle S_{j+1} \rangle$.

Note that Lemma 2.8 gives a description of the generators of $\langle S_j \rangle_{\beta_j}$, which we can use for testing. Furthermore, Lemma 2.9 ensures that we have an SGS for $\langle S_{j+1} \rangle$, so we can test membership by sifting.

If all Schreier generators of $\langle S_j \rangle_{\beta_j}$ are in $\langle S_{j+1} \rangle$ and we thus have verified (2.7), we are up to date above level $j - 1$. Otherwise we have computed a nontrivial siftee h which we add to S_{j+1} and are up to date above level $j + 1$. In the case $j = k$ we also add a new $\beta_{k+1} \in \Omega$ to our list with $h^{\beta_{k+1}} \neq h$.

Algorithm 2.4 depicts a more formal description of the process. The call `Sift` with parameter $j + 1$ means we use Algorithm 2.3 on our partial base $(\beta_{j+1}, \dots, \beta_{|B|})$ and partial strong generating set $\bigcup_{i \geq j+1} S_i$ because we are testing for membership in S_{j+1}. Our Schreier generator sifts through if and only if $k \geq |B| - (j + 1) + 1$ and h is the identity, thus the check in line *14*. In an implementation we could slightly improve this algorithm by avoiding to sift the same Schreier generator twice when we reach the `for`-loop at line *12* at a j we have worked at before.

Another inefficiency occurs when we use this algorithm with Schreier tree transversals. Suppose we construct an orbit α^S and during its construction create an orbit element β^x from some $x \in S$ and a previous element $\beta \in \alpha^S$. Then the Schreier generator $g := u_\beta x u_{\beta^x}^-$ constructed from β and x is always the identity, $g = ()$. We call these Schreier generators **trivial by definition** as coined by [HEO05, Sec. 4.1] and we can ignore them in the Schreier-Sims algorithm. However, we are only able to detect Schreier generators that are trivial by definition if we use a Schreier tree transversal. In this case, constructing a Schreier generator from the pair (β, x), we know that β^x was constructed by x because $\overrightarrow{\beta^x\,\beta}$ is an edge of the tree with label x. In the other case of an explicit transversal we have no history information of how $\beta^x \in \alpha^S$ was initially constructed. Thus we cannot decide

```
    Input: $B_0 = (\beta_1, \dots, \beta_k)$ possibly empty prescribed base, $G = \langle S \rangle \leq \mathrm{Sym}(\Omega)$
    Output: base $B$ and corresponding strong generating set $S$
 1  $B \leftarrow B_0$
 2  if $|B| = 0$ then
 3      find first base point $\beta_1 \in \Omega$ with $\beta_1^S \supsetneq \{\beta_1\}$
 4      $B \leftarrow (\beta_1)$
 5  end
 6  for $i = 1$ to $|B|$ do
 7      $S_i \leftarrow S \cap G_{(\beta_1,\dots,\beta_i)}$
 8      compute orbit $\Delta^{(i)} = \beta_i^{S_i}$ and corresponding transversal $U^{(i)}$
 9  end
10  $j \leftarrow 1$
11  while $j \geq 1$ do
12      forall $\beta \in \Delta^{(j)}, x \in S_j$ do
13          $h, k \leftarrow$ Sift $(u_\beta x u_{\beta^x}^{-}, j+1)$
14          if $k + j < |B|$ or $h \neq ()$ then
15              if $j > |B|$ then
16                  find new base point $\beta_{j+1} \in \Omega$ with $h^{\beta_{j+1}} \neq h$
17                  add $\beta_{j+1}$ to base $B$
18              end
19              $S_j \leftarrow S_j \cup \{h\}$
20              recompute orbit $\Delta^{(j)} = \beta_j^{S_j}$ and corresponding transversal $U^{(j)}$
21              $j \leftarrow j + 1$
22              next $j$ ;                              // i.e. jump to line 11
23          end
24      end
25      $j \leftarrow j - 1$
26  end
27  $S \leftarrow \bigcup_i S_i$
28  return $B, S$
```

Algorithm 2.4: Schreier-Sims BSGS construction

whether a Schreier generator is trivial by definition and have to actually compute it. Not all trivial Schreier generators are trivial by definition, but some are and we can detect this with a Schreier tree and ignore these generators before building the product $u_\beta x u_{\beta^x}^{-}$.

The asymptotic running time behavior of this algorithm depends on how we handle transversals. For our analysis we can split up the algorithm into three parts: setup, orbit and transversal construction and finally sifting. Let $T_{\text{construct}}(X)$ be the time to compute an orbit and the related transversal for the action of a point under a set X. Moreover, let T_{access} be the time to access a transversal element. We further assume that we are not given a prescribed base and compute B from scratch. It will become clear in the course of the analysis how to extend the result for a prescribed base.

In the setup phase we spend $O(T_{\text{construct}}(|S|) + n)$ time to construct a first base point and its orbit. During the main loop, lines *11* through *26*, we add at most $\log G$ elements to each set S_j because every time we do this $|\langle S_j\rangle|$ increases. This sums up to a time of $O(|B|\log|G|T_{\text{construct}}(\log|G|)+\log|G|T_{\text{construct}}(|S|))$.

For the sifting part, we note that we have to consider each of the $\sum_j |U^{(j)}||S_j| \in O(n\log^2|G|+|S|n)$ Schreier generators only once. Sifting can be done in $O(\log|G|T_{\text{access}})$ time. So the sifting part sums up to $O((n\log^2|G|+|S|n)\log|G|T_{\text{access}})$.

We can plug in the results from Section 2.2.1, summarized in Figure 2.4 on page 15, for T_{access} and $T_{\text{construct}}$. For explicit transversals we thus need $O(n^2\log^3|G|+|S|n^2\log|G|)$ time. Choosing Schreier tree transversals results in a $O(n^3\log^3|G|+|S|n^3\log|G|)$ time complexity. Regarding memory usage we note that we always have a $O(|S|n)$ requirement for the group generators. Additionally, we have to store $\sum_j |S_j|$ strong generators which is $O(\log^2|G|)$. By a result of [CST89, Thm. 1], this is also $O(n\log|G|)$, even if $\log|G| \notin O(n)$. For the $\log|G|$ many transversals $U^{(j)}$ we need in the explicit case $O(n^2)$ memory each and $O(n)$ in the Schreier tree case. Thus the memory requirements sum up to $O(n^2\log|G|+|S|n)$ and $O(n\log^2|G|+|S|n)$, respectively.

There are a lot of other BSGS construction algorithms. [But91, Ch. 14] provides a nice overview of the deterministic algorithms before 1990, which are, roughly speaking, also $O(n^5)$. All these deterministic algorithms tend to scale badly for groups of large degree because there may be a lot of Schreier generators to be constructed, which then have to be sifted. A practical alternative are randomized constructions with deterministic or randomized routines checking for the SGS property. The interested reader may find in [Ser03, Sec. 4.5] a nearly linear-time randomized construction algorithm, which is also implemented in GAP. [Ser03, Ch. 8] presents two classical strong generating tests commonly used in GAP and Magma.

2.4 Transversal revisited – shallow Schreier trees

The basic Schreier tree construction algorithm we looked at in Section 2.2.1 results in trees of worst-case depth n. To see this, consider for instance a cyclic group with only one generator: in this case the tree degenerates and every node has at most one child. Using a Schreier tree, the time needed to construct transversal elements, which we have seen to be a fundamental part of algorithms, grows proportionally with the depth of the tree. Shallow trees with a smaller maximal depth can decrease the performance penalty arising from unbalanced, degenerated trees.

There are at least two methods for constructing **shallow Schreier trees** with a $O(\log|G|)$ depth guarantee: a randomized and a deterministic one. A detailed analysis of both the deterministic and the randomized algorithm can be found in [Ser03, Sec. 4.4]. In this section we will only cover the deterministic method originally due to [Bab91]. We begin with the definition of a cube:

Definition 2.10. Let $g_1,\ldots,g_k \in G$. We define the **cube** C_k of $g_1,\ldots,g_k$ as the set $C_k := \{g_1^{\epsilon_1}g_2^{\epsilon_2}\cdots g_k^{\epsilon_k} : \epsilon_1,\ldots,\epsilon_k \in \{0,1\}\}$. We say the cube C_k is **non-degenerate** if $|C_k| = 2^k$ is maximal. Finally, we define the inverse cube $C_k^- := \{g^- : g \in C_k\}$.

We can use the following lemma to actually construct non-degenerate cubes.

Lemma 2.11. Let $g_1,\ldots,g_k,g_{k+1} \in G$ and C_k, C_{k+1} the corresponding cubes of g_1 up to g_k and g_{k+1}, respectively. Then $|C_{k+1}| = 2|C_k|$ if and only if $g_{k+1} \notin C_k^- C_k$.

Proof. $|C_{k+1}| = 2|C_k|$ is equivalent to the fact that the sets C_k and $C_k g_{k+1}$ are disjoint. They are disjoint if and only if $g_{k+1} \notin C_k^- C_k$. □

So for the construction of a non-degenerate cube C_k we iteratively construct cubes C_{i+1} from C_i such that $g_{i+1} \notin C_i^- C_i$. Membership testing in $C_i^- C_i$ in general is difficult (cf. [Ser03, p. 65]), but for our purposes an easier to compute condition for non-membership is sufficient. Clearly, if, for some $\alpha \in \Omega$, we find a $g \in G$ with $\alpha^g \notin \alpha^{C_i^- C_i}$, then $g \notin C_i^- C_i$.

We can compute the set $\alpha^{C_k^- C_k}$ iteratively in $O(kn)$ time. Beginning with $\Delta_1 := \{\alpha\}$, we set $\Delta_{i+1} := \Delta_i \cup \Delta_i^{h_i}$ where h_i is the i-th member of the sequence $g_k^-, g_{k-1}^-, \ldots, g_1^-, g_1, g_2, \ldots, g_k$. This induces a directed, labeled graph on the vertex set $\alpha^{C_k^- C_k}$ with label set $\{h_i : 1 \leq i \leq 2k\} = \{g_i^-, g_i : 1 \leq i \leq k\}$. In this a graph an edge $\overrightarrow{\alpha_1\, \alpha_2}$ exists and has label h_i^-, if $\alpha_1 = \alpha_2^{h_i}$ for some i. A breadth-first-search on this graph yields a Schreier tree as in Definition 2.5.

Suppose we have a non-degenerate cube C_k of $g_1, \ldots, g_k \in G$ with $\alpha^{C_k^- C_k} = \alpha^G$. The non-degeneracy of the cube ensures $k \leq \log|G|$, thus the Schreier tree constructed as above has depth at most $2 \cdot k \leq 2\log|G|$.

It remains to be discussed how we can construct the mentioned cube C_k with $\alpha^{C_k^- C_k} = \alpha^G$. To this end, we find a first element g_1 that moves α. Especially $g_1 \neq ()$, so C_1 is non-degenerate. If $\alpha^{C_1^- C_1} = \alpha^G$ we are done. If this is not the case then we find an $s \in G$ with $\alpha^s \notin \alpha^{C_1^- C_1}$ and set $g_2 := s$. We then have an extended non-degenerate cube C_2 which either suffices to generate the orbit or we find another group element to extend the cube.

Algorithm 2.5 has the details. What makes it rather long compared to the idea described above is to avoid repeating checks for $\alpha^{C_i^- C_i}$ when working on $\alpha^{C_{i+1}^- C_{i+1}}$. Therefore, when we have a new cube element g in line *8*, we first compute $\alpha^{C_i^- C_i g}$ in lines *9* to *16* and then $\alpha^{g^- C_i^- C_{i+1}}$ in lines *18* to *31*. Furthermore, important to note is that we treat L as an ordered set so that we insert g at the end (l. *17*) and g^- at the front (l. *32*) of the label sequence.

Constructing a shallow variant of a Schreier tree costs extra time. As in every orbit computation we always need $O(|S|n)$ by the two `for`-loops in lines *4* and *5* of Algorithm 2.5. Besides that, we also have to deal with cubes. As we have seen before, for every $i \leq k$ the set $\alpha^{C_i^- C_i}$ can be computed in $O(nk)$ time, which we have to construct at most k times. Multiplying $i \leq k$ group elements in lines *7* and *8* also can surely be done in $O(nk)$ time, regardless of how permutations are implemented. Hence, with the bound $k \leq \log|G|$, we get $n\log^2|G|$ additional costs. This results in a total time of $O(n\log^2|G| + |S|n)$. The table in Figure 2.4 summarizes the asymptotic results of this section and Section 2.2.1.

	Explicit	Schreier Tree	Shallow Schreier tree
Construction time	$O(\|S\|n^2)$	$O(\|S\|n)$	$O(n\log^2\|G\| + \|S\|n)$
Memory usage	$O(n^2)$	$O(n + \|S\|n)$	$O(n\log\|G\|)$
Transversal element access	$O(1)$	$O(n^2)$	$O(n\log\|G\|)$

Figure 2.4: Asymptotic time and memory requirements for different transversal implementations

Note that Algorithm 2.5 does not employ a breadth-first-search to build the tree. It still achieves the $2\log|G|$-depth bound because this only depends on the choice of the label set.

```
Input: $G = \langle S \rangle$ permutation group acting on $\Omega$, $\alpha \in \Omega$
Output: $\alpha^G$, labeled edge set $E$ for Schreier tree and label set $L$
$\Delta \leftarrow \{\alpha\}$
$E \leftarrow \emptyset$
$L \leftarrow \emptyset$
for $\beta \in \Delta$ do
    for $s \in S$ do
        if $\beta^s \notin \Delta$ then
            $u_\beta \leftarrow$ multiplied edge labels along path from $\beta$ to $\alpha$
            $g \leftarrow u_\beta s$ ;                          // our new cube element
            // extend orbit by $\Delta^g$
            $\Gamma \leftarrow \emptyset$
            for $\gamma \in \Delta$ do
                if $\gamma^g \notin \Delta$ then
                    $\Gamma \leftarrow \Gamma \cup \{\gamma^g\}$
                    add edge $\overrightarrow{\gamma^g\,\gamma}$ with label $g^-$ to $E$
                end
            end
            $\Delta \leftarrow \Delta \cup \Gamma$
            $L \leftarrow L \cup \{g\}$
            // check the extension of $\alpha$ with $g^-$
            if $\alpha^{g^-} \notin \Delta$ then
                $\Delta \leftarrow \Delta \cup \{\alpha^{g^-}\}$
                add edge $\overrightarrow{\alpha^{g^-}\,\alpha}$ with label $g$ to $E$
            end
            // check all possible extensions of $\alpha^{g^-}$ with the current
               labels
            $\Gamma \leftarrow \{\alpha^{g^-}\}$
            for $l \in L$ do
                for $\gamma \in \Gamma$ do
                    if $\gamma^l \notin \Delta$ then
                        $\Gamma \leftarrow \Gamma \cup \{\gamma^l\}$
                        add edge $\overrightarrow{\gamma^l\,\gamma}$ with label $l^-$ to $E$
                    end
                end
            end
            $\Delta \leftarrow \Delta \cup \Gamma$
            $L \leftarrow \{g^-\} \cup L$
        end
    end
end
return $\Delta, E, L$
```

Algorithm 2.5: Orbit and shallow Schreier tree

A breadth-first-approach, however, will produce trees of lesser or equal depth on the same input. For an implementation of this improvement we would have to keep track of the depth $d(\alpha)$ of orbit elements $\alpha \in \Delta$ in the already constructed part of the tree. A bucket sort (cf. [CLRS09, Sec. 8.4]) with respect to $d(\alpha)$, providing insertion into a $d(\alpha)$-sorted sequence in $O(1)$ time, could be used without deteriorating the asymptotic time behavior given above.

2.5 Base change

As we will see later, many applications that rely on computations with bases and strong generating sets require a specific base. For example, if we want to compute stabilizers (cf. Section 3.1.4) then we need a base which is closely related to the set we like to stabilize. Because constructing a complete BSGS from scratch is likely to be an expensive process we would like to have an algorithm that modifies a strong generating set with respect to base $(\beta_1, \dots, \beta_m)$ so that it is strong generating relative to some other base $(\alpha_1, \dots, \alpha_k)$. In this section we will examine several solutions to this problem. The interested reader may also find several of the presented algorithms with examples in [But91, Ch. 12] and [Ser03, Sec. 5.4].

Assume for a moment that we know how to perform a base point transposition. Given an SGS relative to a base $(\beta_1, \dots, \beta_m)$, a **base point transposition** at i constructs an SGS relative to $(\beta_1, \dots, \beta_{i-1}, \beta_{i+1}, \beta_i, \beta_{i+2}, \beta_{i+3}, \dots, \beta_m)$. We can use this method to insert a new base point $\alpha \in \Omega$ at a specific position $j \leq m$. First we insert α as a redundant base point after the first β_i such that $G^{[i+1]} = G_{(\beta_1,\dots,\beta_i)}$ fixes α. This ensures that $G_{(\beta_1,\dots,\beta_i)} = G_{(\beta_1,\dots,\beta_i,\alpha)}$ and the new transversal that we insert is trivial. Then we apply $|i - j|$ many transpositions to bring α into the desired position, which gives us a possibly redundant base $(\beta_1, \dots, \beta_{j-1}, \alpha, \beta_j, \dots, \beta_m)$. Note that if $i < j$ all necessary transpositions are trivial and we can also insert α directly at position j. By repeating such insertions with α_1 at position 1, α_2 at position 2 &c., and stripping redundant elements afterward, we can change the base to a completely different one $(\alpha_1, \alpha_2, \dots, \alpha_{k-1}, \alpha_k, \dots)$.

Algorithm 2.6 formalizes this base change procedure. The following two sections present two algorithms, one deterministic, one randomized, to perform a base point transposition.

2.5.1 Deterministic base point transposition

The first, deterministic transposition algorithm is due to Sims (cf. [Sim71b]). Let S be an SGS for G relative to the base $(\beta_1, \dots, \beta_m)$ with stabilizer chain $G = G^{[1]} \geq \cdots \geq G^{[m]} \geq G^{[m+1]} = 1$ and transversals $U^{(1)}, \dots, U^{(m)}$. Our goal is to compute an SGS $\bar{S}$ relative to $(\beta_1, \dots, \beta_{i-1}, \beta_{i+1}, \beta_i, \dots, \beta_m)$ with stabilizer chain $G = \bar{G}^{[1]} \geq \cdots \geq \bar{G}^{[m]} \geq \bar{G}^{[m+1]} = 1$ and transversals $\bar{U}^{(1)}, \dots, \bar{U}^{(m)}$.

Most of the stabilizers do not change, we have $\bar{G}^{[j]} = G^{[j]}$ for every $j \neq i + 1$. Accordingly for $1 \leq j < i$ and $i + 1 < j \leq m$ the transversals do not change, $\bar{U}^{(j)} = U^{(j)}$. We can easily compute the new i-th fundamental orbit $\beta_{i+1}^{G^{[i]}}$ and obtain a new transversal $\bar{U}^{(i)}$. It remains to set up $\bar{U}^{(i+1)}$.

As we do not know a generating set for $\bar{G}^{[i+1]}$ yet we iteratively construct the new $(i + 1)$-st fundamental orbit $\bar{\Delta}^{(i+1)}$, starting with $\bar{\Delta}^{(i+1)} := \{\beta_i\}$ and $\bar{S} := S$. Then we

```
Input: $B = (\beta_1, \dots, \beta_m)$, $S$ BSGS, sequence of new base points $(\alpha_1, \dots, \alpha_k)$
Output: updated BSGS with base $(\alpha_1, \dots, \alpha_k, \dots)$
for $i = 1$ to $k$ do
    if $\beta_i \neq \alpha_i$ then
        // find insertion position
        $j \leftarrow i + 1$
        while $\alpha_i^{G^{[j+1]}} \supsetneq \{\alpha_i\}$ do
            $j \leftarrow j + 1$
        end
        insert $\alpha_i$ into $B$ at position $j$
        insert $\Delta^{(i)} = \{\alpha_i\}$ and corresponding transversal $U^{(i)}$ into sequence of orbits
        and transversals at position $j$
        while $j > i$ do
            $B, S \leftarrow$ Transpose $(B, S, j - 1)$
            $j \leftarrow j - 1$
        end
    end
end
```

Algorithm 2.6: Base change with transpositions

construct Schreier generators from $\bar{U}^{(i)}$ and $\bar{G}^{[i]}$. Whenever we find a Schreier generator $h \in \bar{G}^{[i+1]}$ that extends our orbit, we add h to $\bar{S}$. With the new generator we recompute $\bar{\Delta}^{(i+1)} := \beta_i^{\langle \bar{S} \cap \bar{G}^{[i+1]} \rangle} = \beta_i^{\langle \bar{S} \cap G^{[i]}_{\beta_{i+1}} \rangle}$. From $|G| = \prod_{i=1}^{m} |U^{(i)}|$ by (2.3) we know that for the final orbit we must have

$$|\bar{\Delta}^{(i+1)}| = |\bar{U}^{(i+1)}| = \frac{|U^{(i+1)}| \cdot |U^{(i)}|}{|\bar{U}^{(i)}|}.$$

If the number of elements in $\bar{\Delta}^{(i+1)}$ is below that number, we proceed with another Schreier generator. In the case we have reached this cardinality for $\bar{\Delta}^{(i+1)}$, we know that we have a complete generating set for $\bar{G}^{[i+1]}$ and, along with our orbit calculations, a transversal $\bar{U}^{(i+1)}$ and so we are done. As in the Schreier-Sims algorithm in Section 2.3, for performance reasons we should avoid constructing Schreier generators that are trivial by definition.

Algorithm 2.7 shows an in-place variant of the transposition part. Again, like in the Schreier-Sims construction, it would be an improvement for an implementation to construct and check each Schreier generator in line *6* only once. A detailed analysis in [But91, p. 123] shows that this deterministic base point transposition has a $O(n^4)$ complexity.

2.5.2 Randomized base point transposition

Random group elements

Instead of computing Schreier generators for $\bar{G}^{[i+1]}$ explicitly, we can use randomness. But first of all we need to know how to efficiently construct random elements of a group.

```
Input: $B = (\beta_1, \dots, \beta_m)$, $S$ BSGS with transversals $U^{(j)}$, $i$ base point index to
       change
Output: updated BSGS with base $(\beta_1, \dots, \beta_{i-1}, \beta_{i+1}, \beta_i, \beta_{i+2}, \dots, \beta_m)$
swap the values of $\beta_i$ and $\beta_{i+1}$
compute orbit $\bar\Delta^{(i)} = \beta_i^{G^{[i]}}$ and corresponding transversal $\bar U^{(i)}$
$\bar S \leftarrow S$
compute orbit $\bar\Delta^{(i+1)} = \beta_{i+1}^{\langle \bar S \cap G^{[i]}_{\beta_i} \rangle}$ and corresponding transversal $\bar U^{(i+1)}$
while $|\bar\Delta^{(i+1)}| < \frac{|U^{(i+1)}| \cdot |U^{(i)}|}{|\bar U^{(i)}|}$ do
    forall $\beta \in \bar\Delta^{(i)}$, $x \in \bar S \cap G^{[i]}$ do
        $h \leftarrow u_\beta x u^-_{\beta^x}$
        if $h^{\beta_{i+1}} \notin \bar\Delta^{(i+1)}$ then
            $\bar S \leftarrow \bar S \cup \{h\}$
            recompute orbit $\bar\Delta^{(i+1)} = \beta_{i+1}^{\langle \bar S \cap G^{[i]}_{\beta_i} \rangle}$ and corresponding transversal $\bar U^{(i+1)}$
            break
        end
    end
end
$S \leftarrow \bar S$
$U^{(i)} \leftarrow \bar U^{(i)}$
$U^{(i+1)} \leftarrow \bar U^{(i+1)}$
$\Delta^{(i)} \leftarrow \bar\Delta^{(i)}$
$\Delta^{(i+1)} \leftarrow \bar\Delta^{(i+1)}$
```

Algorithm 2.7: Base point transposition – deterministic version

Because we shall need the uniform random distribution of elements several times we write in short $\mathcal{U}(G)$ for the uniform distribution on the set or group G. We assume throughout the chapter that we have means to generate uniformly distributed elements from a given set.

Efficiently generating $g \in \mathcal{U}(G)$ for a general group G is a hard task, which is not covered by this thesis. Both [Ser03, Sec. 2.2] and [HEO05, Sec. 3.2.2] contain algorithm descriptions for that. The situation is much easier if we have a base and strong generating set for G. Equation (2.2) tells us that every $g \in G$ is uniquely composed from transversal elements

$$g = u_m u_{m-1} \cdots u_2 u_1, \quad \text{for some } u_i \in U^{(i)}.$$

Hence, if we choose $u_i \in \mathcal{U}(U^{(i)})$ for all i we get a uniformly distributed group element g as the product of the u_i.

When we have access to $\mathcal{U}(G)$ and a transversal U for G mod G_α, $\alpha \in \Omega$, we also can compute uniformly distributed random elements of the stabilizer G_α as follows:

Lemma 2.12. Let $\alpha \in \Omega$ and U be a transversal for G mod G_α. Further, let $g \in \mathcal{U}(G)$ and $\beta := \alpha^g$. Consider $u_\beta \in U$ such that $\alpha^{u_\beta} = \beta$. Then $g u_\beta^-$ is uniformly distributed in G_α.

Proof. Let $\{u_1, \dots, u_k\} =: U$ be the transversal elements. Then G can be partitioned in $G = \bigcup_{i=1}^{k} G_\alpha u_i$. Every $g \in G$ is in exactly one coset $G_\alpha u_\beta$. So every coset $G_\alpha u_i$ has the same chance to occur as $G_\alpha u_\beta$ and every $h \in G_\alpha$ has the same chance to occur as $g u_\beta^-$. Thus $g u_\beta^-$ is uniformly distributed in G_α if g is uniformly distributed in G. □

Transposition

For a base transposition we are in the situation that we know $\bar{G}^{[i]}$ and have a transversal $\bar{U}^{(i)}$ for $\bar{G}^{[i]}$ mod $\bar{G}^{[i+1]}$. We then need generators for $\bar{G}^{[i+1]} = \bar{G}^{[i]}_{\beta_{i+1}}$. So instead of computing Schreier generators, we generate $g \in \mathcal{U}(\bar{G}^{[i+1]})$ according to Lemma 2.12 from $\bar{U}^{(i)}$ and $\bar{G}^{[i]} = G^{[i]}$, which we already have an SGS for. Then we test whether g extends the known transversal part. If it does then we add it to our generating set and start over again.

```
Input: $B = (\beta_1, \dots, \beta_m)$, $S$ BSGS with transversals $U^{(j)}$, $i$ base point index to change
Output: updated BSGS with base $(\beta_1, \dots, \beta_{i-1}, \beta_{i+1}, \beta_i, \beta_{i+2}, \dots, \beta_m)$
swap the values of $\beta_i$ and $\beta_{i+1}$
compute orbit $\bar{\Delta}^{(i)} = \beta_i^{G^{[i]}}$ and corresponding transversal $\bar{U}^{(i)}$
$\bar{S} \leftarrow S$
compute orbit $\bar{\Delta}^{(i+1)} = \beta_{i+1}^{\langle \bar{S} \cap G^{[i]}_{\beta_i} \rangle}$ and corresponding transversal $\bar{U}^{(i+1)}$
while $|\bar{\Delta}^{(i+1)}| < \frac{|U^{(i+1)}| \cdot |U^{(i)}|}{|\bar{U}^{(i)}|}$ do
    generate $g \in \mathcal{U}(G^{[i]})$
    find $u_g \in \bar{U}^{(i)}$ with $\beta_{i+1}^{u_g} = \beta_{i+1}^{g}$
    $h \leftarrow g u_g^-$
    if $h^{\beta_{i+1}} \notin \bar{\Delta}^{(i+1)}$ then
        $\bar{S} \leftarrow \bar{S} \cup \{h\}$
        recompute orbit $\bar{\Delta}^{(i+1)} = \beta_{i+1}^{\langle \bar{S} \cap G^{[i]}_{\beta_i} \rangle}$ and corresponding transversal $\bar{U}^{(i+1)}$
    end
end
$S \leftarrow \bar{S}$
$U^{(i)} \leftarrow \bar{U}^{(i)}$
$U^{(i+1)} \leftarrow \bar{U}^{(i+1)}$
$\Delta^{(i)} \leftarrow \bar{\Delta}^{(i)}$
$\Delta^{(i+1)} \leftarrow \bar{\Delta}^{(i+1)}$
```

Algorithm 2.8: Base point transposition – randomized version

One can prove that, if $\bar{S}$ is not complete then with probability of at least $\frac{1}{2}$ such randomly generated g will not sift through our partial BSGS and therefore extend our generating set (cf. [Ser03, Lemma 4.3.1]). Also note that because we know the size of the new transversal $\bar{U}^{(i+1)}$ in advance we know when we have found enough generators to make $\bar{S}$ a strong generating set. Hence we have a randomized base transposition algorithm and we can tell whether its result is correct by looking at the size of the transversal product.

Algorithm 2.8 formalizes the process and is very similar to the deterministic version in Algorithm 2.7. The only difference is how we find the probable generators h. Note that because of the randomized nature of this algorithm we cannot guarantee its termination.

2.5.3 Base change by conjugation

Now that we know all the details of changing a base with Algorithm 2.6 we have a second look at it and reduce the number of transpositions that we need to apply to $B := (\beta_1, \ldots, \beta_m)$ so that it is prefixed by $(\alpha_1, \ldots, \alpha_k)$. One way to accomplish that is conjugation as observed by [Sim71a]. If we have a BSGS (B, S) then we can easily construct another valid BSGS (B^g, S^g) for $g \in G$. Here $B^g = (\beta_1^g, \ldots, \beta_m^g)$ as usual and $S^g := \{g^- sg \,:\, s \in S\}$ denotes group conjugation. To turn this rather special case in a generally applicable algorithm we use conjugation in combination with transpositions.

Let us assume we have a $g \in G$ which we can use for the first $j-1$ elements, i. e. we know a base $B' := (\gamma_1, \ldots, \gamma_l)$ and g such that

$$\gamma_i^g = \alpha_i \quad \text{for } 1 \leq i \leq j-1. \tag{2.10}$$

At the beginning we could start with $g = ()$ and $j = 1$. If $\alpha_j^{g^-}$ is in the j-th basic orbit (with respect to B') we find a $u \in U'^{(j)}$ with $\gamma_j^u = \alpha_j^{g^-}$ and update $g \leftarrow ug$. Because $u \in U'^{(j)}$ and thus acts as identity on the first $j-1$ base elements, property (2.10) is preserved and extends to j. In the other case that $\alpha_j^{g^-}$ is not in the j-th basic orbit we insert $\alpha_j^{g^-}$ (via repeated transpositions) at position j, resulting in $\gamma_j = \alpha_j^{g^-}$. After this step we have a g fulfilling (2.10) for the first j elements and we can continue this way until we have covered all elements at $j = k$. Algorithm 2.9 modifies Algorithm 2.6 such that it uses conjugation where applicable.

2.5.4 Base change by construction – randomized BSGS construction

Both presented base change algorithms so far may be slow when a long new base sequence is prescribed. In such cases it would be an alternative to compute a new BSGS from scratch to avoid a high number of transpositions. In this section we look at a randomized version of the Schreier-Sims construction, which we may choose to compute a new base and strong generating set if the prescribed base differs substantially from the existing one and we expect too many transpositions. However, in practice it may be difficult to recognize automatically in advance with certainty whether computation from scratch is faster than transpositions.

We randomize the Schreier-Sims algorithm 2.4 in a similar way as the randomized transposition algorithm 2.8 originates from its deterministic counterpart Algorithm 2.7. The bottleneck of the Schreier-Sims construction is that all Schreier generators have to be sifted to ensure that the generating set is also a strong generating set by Lemma 2.9. We now relax this criterion and use random generators. To make the construction work without errors we have to know the group order in advance. This is always the case when we want to perform a base change because we can read it from the existing transversal system (2.3).

```
Input: $B = (\beta_1, \dots, \beta_m), S$ BSGS, sequence of new base points $(\alpha_1, \dots, \alpha_k)$
Output: updated BSGS with base $(\alpha_1, \dots, \alpha_k, \dots)$
$g \leftarrow ()$;
for $i = 1$ to $k$ do
    $\alpha_i \leftarrow \alpha_i^{g^-}$;
    if $\beta_i \neq \alpha_i$ then
        if $\alpha_i \in \Delta^{(i)}$ then
            find element $u_{\alpha_i} \in U^{(i)}$ such that $\beta_i^{u_{\alpha_i}} = \alpha_i$;
            $g \leftarrow u_{\alpha_i} g$;
        else
            // find insertion position
            $j \leftarrow i + 1$;
            while $\alpha_i^{G^{[j+1]}} \supsetneq \{\alpha_i\}$ do
                $j \leftarrow j + 1$;
            end
            insert $\alpha_i$ into $B$ as position $j$;
            insert $\Delta^{(i)} = \{\alpha_i\}$ and corresponding transversal $U^{(i)}$ into sequence of
            orbits and transversals;
            while $j > i$ do
                $B, S \leftarrow$ Transpose $(B, S, j - 1)$;
                $j \leftarrow j - 1$;
            end
        end
    end
end
$B \leftarrow B^g$;
$S \leftarrow S^g$;
for $i = 1$ to $k$ do
    recompute orbit $\Delta^{(i)}$ and corresponding transversal $U^{(i)}$;
end
```

Algorithm 2.9: Base change with conjugation and transpositions

To construct a base and strong generating set B, S in a randomized fashion for a group $G = \langle X \rangle$, we start similarly to the deterministic Schreier-Sims algorithm. At the beginning we choose a $\beta_1 \in \Omega$ which is moved by at least one generator and start with $B = (\beta_1)$ and $S_1 := X$. If we have a prescribed base we of course skip this step. Let us assume we already have constructed some subsets $S_1, \dots, S_k$, generating subsets of $G^{[1]}, \dots, G^{[k]}$.

We generate uniformly independently distributed random elements $g \in \mathcal{U}(G)$ according to the results of Section 2.5.2. We then sift them through the existing transversal system. If one of these g does not sift through we end up with an index j and a siftee h. We can add this siftee h to S_j to improve our approximation of a strong generating set. If $j = k$ we also add a new base point β_{k+1}. When the product $\prod_{i=1}^{k} |U^{(i)}|$ equals the known base order we are done because no element $g \in G$ can have a non-trivial siftee without enlarging the transversal product. If the orders mismatch we start again sifting random elements through an augmented set system $S_1, \ldots, S_k$. Algorithm 2.10 formalizes this procedure.

Input: $B = (\beta_1, \ldots, \beta_k)$ prescribed base, $G = \langle X \rangle \leq \mathrm{Sym}(\Omega)$, a random generator for $\mathcal{U}(G)$, group order $|G|$
Output: base B and corresponding strong generating set S

```
if $|B| = 0$ then
    find first base point $\beta_1 \in \Omega$ with $\beta_1^X \supsetneq \{\beta_1\}$;
    $B \leftarrow (\beta_1)$;
end
for $i = 1$ to $|B|$ do
    $S_i \leftarrow X \cap G_{(\beta_1,\ldots,\beta_i)}$;
    compute orbit $\Delta^{(i)} = \beta_i^{S_i}$ and corresponding transversal $U^{(i)}$;
end
repeat
    compute $g \in \mathcal{U}(G)$;
    $h, j \leftarrow$ Sift $(g)$;
    if $h \neq ()$ then
        if $j \geq |B|$ then
            find new base point $\beta_j \in \Omega$ with $h^{\beta_j} \neq h$;
            add $\beta_j$ to base $B$;
        end
        $S_j \leftarrow S_j \cup \{h\}$;
        recompute orbit $\Delta^{(j)} = \beta_j^{S_j}$ and corresponding transversal $U^{(j)}$;
    end
until $\prod_{i=1}^{k} |U^{(i)}| = |G|$ ;
$S \leftarrow \bigcup_i S_i$
return $B, S$
```

Algorithm 2.10: Randomized Schreier-Sims BSGS construction

If the BSGS of a group has to be changed to a significantly different base, using this algorithm to compute a new BSGS from scratch may be the fastest way. We will come back to this in Section 4.2.3 when we look at results from experiments with different base change algorithms.

One could also use this algorithm if an initial BSGS has to be constructed and no prior BSGS is known. The first problem then is to generate random group elements because we do not have access to $\mathcal{U}(G)$. A second problem may be that the group order is unknown in advance and we thus do not know when the construction is complete. There is an algorithm often referred to as “random Schreier-Sims algorithm” by Leon (cf. [Leo80]), which addresses both problems heuristically. Though there is not much to modify in Algorithm 2.10 to adapt these solutions, this is beyond the scope of this thesis. The interested reader may also consult [Ser03, Sec. 4.3] for further explanations concerning this approach.

3 Backtrack Search

An often occurring problem when dealing with symmetries is to compute the setwise stabilizer G_Δ for a subset $\Delta \subseteq \Omega$ under action of $G \leq \mathrm{Sym}(\Omega)$. This problem naturally arises during symmetry computation when a computation is restricted to a subproblem defined by $\Delta \subset \Omega$. For example, consider the left polyhedral cone in Figure 3.1. The reader not too familiar with the terminology of polyhedrons may consult [Zie95] for definitions.

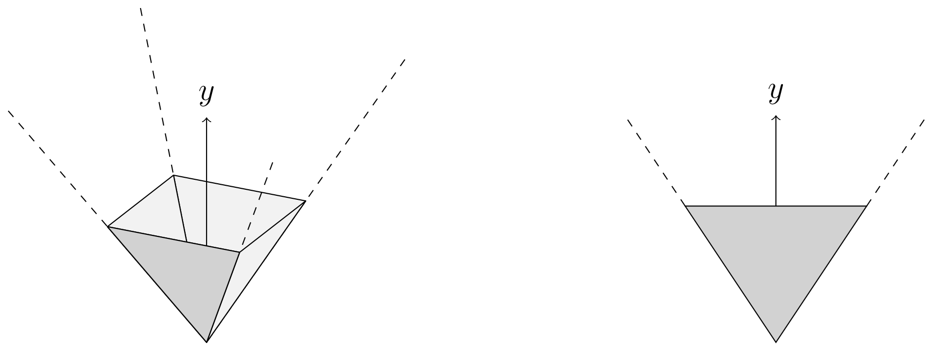


Figure 3.1: Polyhedral cones

Suppose we were given the four half-spaces defining the cone, identified with $\{1, 2, 3, 4\}$, and our goal was to compute the vertices and rays of the cone up to its symmetry. The symmetry group G of the left cone is the same as that of the square: rotations around the y-axis by multiples of 90 degree and four reflections, the dihedral group D_4 of order 8. If the left cone was too hard to work with we could branch into a lower-dimensional subproblem first, depicted on the right. The suitable symmetry group to use for this would be the stabilizer G_1 of the corresponding half-space 1 in G: one horizontal reflection and the identity. In this two-dimensional cone we would then find one vertex (the apex of the cone) and one ray up to symmetry because the second ray can be obtained from the first by reflection. Going back to the full three-dimensional cone and solving the other three remaining subproblems would give us one apex and four candidate rays in total. From these four rays we need of course only one because the others can be obtained by rotation. For more elaborate ways to compute vertices and rays of polyhedrons up to symmetries the interested reader may look into [BSS09].

A second interesting problem in symmetry computation is group intersections. For instance, consider solving linear programs up to symmetries. In this case we could compute the symmetry group of the whole problem as intersection of the symmetry group of the polyhedron we are optimizing over and the symmetry of the target function.

Both the set stabilizer and group intersection problem have been shown to be polynomial-time equivalent to the problem of computing graph isomorphisms. Thus no general, provable theoretically fast algorithms are known. In practice, however, graph isomorphisms can often be computed efficiently, despite the unknown complexity class membership in P (cf. [Luk93]). Like graph isomorphisms, those group problems are usually solved with a backtracking approach. An algorithm walks through the group and finds all elements with the desired properties, in this case generating a set stabilizer or a group intersection. In this section we will study two BSGS-based algorithms for this task, one of them inspired by graph isomorphism techniques. These backtracking algorithms are fairly general and also commonly used for other group-theoretic problems like centralizers and normalizers, which we exclude from our considerations here.

In a general setting, we attempt to find the set $G(\mathcal{P})$ of all elements of a group G satisfying a mathematical property $\mathcal{P}$. It seems reasonable that for "unstructured" properties like an arbitrary equation in group elements the search has to "touch" almost every single group element. One kind of structure that helps in bounding the search space is that $G(\mathcal{P})$ is a subgroup of G or a coset of a subgroup (or empty). In this case search problems usually become tractable despite a worst-case complexity of $\Omega(|G|)$.

We can also use these subgroup search methods to compute automorphisms of combinatorial objects such as matrices. For instance, for a matrix A we may set $G := \mathrm{Sym}(A)$ as the group of all permutations acting on A and define $\mathcal{P}$ to be to true for a $g \in G$ if and only $A^g = A$. Then $G(\mathcal{P})$ is the automorphisms group of all matrices. To compute $G(\mathcal{P})$ the backtrack framework may remain the same in general, only $\mathcal{P}$ specific components have to be adapted.

For every subgroup problem there is a related coset problem where we would like to find one representative of the coset. For instance, consider the set stabilizer problem. The subgroup task is to find the setwise stabilizer G_Δ for a set $\Delta \subseteq \Omega$. The corresponding coset problem is, given $\Gamma, \Delta \subseteq \Omega$, to find $g \in G$ with $\Delta^g = \Gamma$ (or establish that no such element exists). In other words, we have to find a representative of a coset $(G_\Delta)g$ with $\Delta^{(G_\Delta)g} = \Gamma$ or a representative of a coset $g(G_\Gamma)$ with $\Delta^{g(G_\Gamma)} = \Gamma$, if such a coset exists. We could then solve the corresponding subgroup problem to obtain the full coset if it is required, but usually one representative is enough. Going back to the example of a polyhedral cone in Figure 3.1, we need to solve such a coset problem if we want to know whether two vertices or rays Γ, Δ can be obtained from each other by action of the symmetry group G.

To illustrate the methods and techniques we will focus on subgroup search. At the end of each section we will discuss what changes are necessary to perform a search for coset representatives.

3.1 Classical backtracking

We organize the search for $G(\mathcal{P})$ based on the concept of a search tree which has G as root and all elements of G as leaves. We traverse this tree depth-first in search for elements fulfilling the property $\mathcal{P}$. While doing that, we hope that we can prune whole subtrees because they contain irrelevant data to construct $G(\mathcal{P})$. Our goal is to examine as few leaves (group elements) as possible.

3.1.1 Search tree

A base together with a strong generating set enables us to enumerate all group elements by enumerating all possible transversal combinations

$$u_n u_{n-1} \cdots u_2 u_1 \quad \text{with } u_i \in U^{(i)}, \tag{3.1}$$

which is our equation (2.2) from page 6. During this enumeration we can check every $u_n u_{n-1} \cdots u_2 u_1$ if it is in $G(\mathcal{P})$. The key to fast results is a clever organization of the search. Thus we set up our search tree in the following way.

Definition 3.1. Let $B = (\beta_1, \ldots, \beta_m)$ be a base for G with strong generating set S and corresponding transversals $U^{(i)}$ for $1 \leq i \leq m$. Our **search tree** is a labeled tree with the following properties:

- The root at level 0 has the empty label $()$.
- We label every node at level $i > 0$ with a sequence $(\gamma_1, \ldots, \gamma_i) \in \Omega^i$ for $1 \leq i \leq m$.
- A node $(\gamma_1, \ldots, \gamma_i)$ at level $i < m$ has the children $(\gamma_1, \ldots, \gamma_i, \gamma_{i+1})$ for each $\gamma_{i+1} \in (\Delta^{(i+1)})^g$ where $g \in G$ is an arbitrary permutation fulfilling $(\beta_1, \ldots, \beta_i)^g = (\gamma_1, \ldots, \gamma_i)$.

This is just a reformulation of the enumeration based on (3.1). We observe that from one node of the search tree to one of its children we fix one more base point image. The parent node has $(\beta_1, \ldots, \beta_i)^g = (\gamma_1, \ldots, \gamma_i)$ for some $g \in G$, the child node has $(\beta_1, \ldots, \beta_i, \beta_{i+1})^h = (\gamma_1, \ldots, \gamma_i, \gamma_{i+1})$ for some $h \in H$. We apply the decomposition (3.1) to $g = u_n \cdots u_1$ and $h = v_n \cdots v_1$ where $u_j, v_j \in U^{(j)}$. Because of the i fixed first base points we must have that $u_j = v_j$ for all $1 \leq j \leq i$. So each path from the root to a leaf represents a sequence of group elements (u_1), $(u_2 u_1)$, $(u_3 u_2 u_1)$, …, $(u_n u_{n-1} \cdots u_2 u_1)$.

This also means that each tree node at level i represents a coset $G^{[i+1]} g$ of G. Although we label the tree with partial base images it is sometimes more convenient to look at a node as a coset, so we will use the notation that is best in the context. More formally, for a node $n := (\gamma_1, \ldots, \gamma_i)$ we define the function coset to return the corresponding coset: $\operatorname{coset}(n) = \{g \in G : \beta_j^g = \gamma_j \ \forall 1 \leq j \leq i\}$. The leaves, $(\gamma_1, \ldots, \gamma_m) = (\beta_1, \ldots, \beta_m)^{u_m u_{m-1} \cdots u_1}$ for some $u_j \in U^{(j)}$, correspond uniquely to group elements $u_m u_{m-1} \cdots u_1 \in G$. Conversely, because of the way we define child nodes, every $g = u_m u_{m-1} \cdots u_1 \in G$ corresponds to a leaf $(\gamma_1, \ldots, \gamma_m) = (\beta_1, \ldots, \beta_m)^g$.

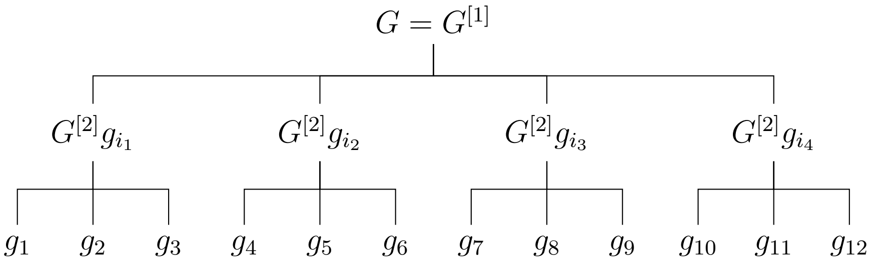


Figure 3.2: Group with twelve elements split up into cosets by a search tree

Example 3.2. Figure 3.2 depicts an example of a group $G = \{g_1, \dots, g_{12}\}$ which is split up into cosets by a search tree. This is also how a search tree for the alternating group A_4 looks like because $|U^{(1)}| = 4$ and $|U^{(2)}| = 3$ for A_4, as a simple calculation shows.

For a cyclic group the search tree degenerates. From Example 2.4 on page 7 we know that for a cyclic group $|U^{(1)}| = |\Omega| = |G|$ regardless of the base. This means that the tree has all its $|G|$ leaves already at level 1, which makes pruning more difficult.

```
Input: BSGS for group $G$ with transversals $U^{(i)}$ and basic orbits $\Delta^{(i)}$, node $n$,
       corresponding $g \in \mathrm{coset}(n)$, level $i$
if $i = m + 1$ then
    VisitGroupElement ($g$)
    return
end
forall $\delta \in \Delta^{(i)}$ do
    $\gamma_{i+1} \leftarrow \delta^g$
    find $u_\delta \in U^{(i)}$ with $\beta_i^{u_\delta} = \delta$
    $n' \leftarrow$ extend $n$ by $\gamma_{i+1}$
    Traverse($n', u_\delta g, i + 1$)
end
```

Algorithm 3.1: Depth-first traversal of the search tree of a group

Algorithm 3.1 displays an exemplary depth-first traversal of a search tree according to Definition 3.1 and provides a basis for the more advanced algorithms to come. We refer to this algorithm `Traverse` for recursion and we start it with the arguments $n = (), g = (), i = 1$. In method `VisitGroupElement` we could implement whatever code we want to be executed for every group element $g \in G$.

When we look for elements of $G(\mathcal{P})$ in our search tree we would like to establish as early as possible that we can skip the traversal of subtrees because the corresponding cosets $G^{[i]}g$ contain irrelevant data. In the following sections we will see how to perform the search for $G(\mathcal{P})$ and how to prune the tree. Before we look at methods that depend on the property $\mathcal{P}$ we examine generally applicable ideas.

3.1.2 Pruning the tree

For our backtrack search we perform a depth-first traversal of our tree defined in Definition 3.1. The following lemma suggests that it is advantageous to find all members of $G^{[i]} \cap G(\mathcal{P})$ before touching any element from $G \setminus G^{[i]}$.

Lemma 3.3. Let $G(\mathcal{P})$ be a subgroup of G and suppose that for some i all elements of $K = G^{[i]} \cap G(\mathcal{P})$ are known. Then for every $h, h' \in G(\mathcal{P})$ with $G^{[i]}h = G^{[i]}h'$ we have that $\langle K, h\rangle = \langle K, h'\rangle$.

Proof. It suffices to show that $Kh = Kh'$. From $G^{[i]}h = G^{[i]}h'$ we know that there is a $u \in G^{[i]} \cap G(\mathcal{P})$ such that $h = uh'$. By definition of K we have $u \in K$ and thus h and h' are in the same coset $Kh = Kh'$. □

If we know $G^{[i]} \cap G(\mathcal{P})$ and we find an $h \in G(\mathcal{P})$ then we can skip the whole rest of its coset $G^{[i]}h$ because we cannot obtain new group generators this way. In the case that $G(\mathcal{P})$ is a coset of a subgroup this lemma is not really relevant because we stop the search as soon as we have found one element.

Another benefit of computing $G^{[i]} \cap G(\mathcal{P})$ first is that we obtain automatically a strong generating set relative to the original base B if $G(\mathcal{P})$ is a subgroup. To see this we observe that $G(\mathcal{P})^{[i]} = G(\mathcal{P}) \cap G^{[i]}$ because $G(\mathcal{P}) \subseteq G$. Hence by examining $G(\mathcal{P}) \cap G^{[i]}$ first we get generators for all subgroups along the stabilizer chain of $G(\mathcal{P})$.

In order to compute $G(\mathcal{P}) \cap G^{[i]}$ before $G \setminus G^{[i]}$ we visit children of the nodes in the search tree in a special order. We introduce an ordering $\prec$ of Ω in which the base elements come first, and in order. For $1 \leq i < j \leq m$ we define $\beta_i \prec \beta_j$ and $\beta_i, \beta_j \prec \alpha$ for all $\alpha \in \Omega \setminus B$. We may set the relationship among $\alpha \in \Omega \setminus B$ arbitrarily to make $\prec$ a complete order. During the search tree traversal we order the children $(\gamma_1, \ldots, \gamma_{i-1}, \gamma_i^{(1)}), \ldots, (\gamma_1, \ldots, \gamma_{i-1}, \gamma_i^{(k)})$ of a node such that $\gamma_i^{(j)} \prec \gamma_i^{(j+1)}$ for all $1 \leq j \leq k-1$. So the first child of $(\gamma_1, \ldots, \gamma_{i-1})$ is always $(\gamma_1, \ldots, \gamma_{i-1}, \beta_i)$. We thus visit all elements of $G^{[i]}$ before any from $G \setminus G^{[i]}$.

```
Input: BSGS for group $G$ with transversals $U^{(i)}$ and basic orbits $\Delta^{(i)}$, node $n$,
       corresponding $g \in \mathrm{coset}(n)$, level $i$, completed level $i_{\text{completed}}$ (global variable)
Output: $K$ generating set of $\langle K \rangle = G(\mathcal{P})$ subgroup of $G$ with property $\mathcal{P}$ if $i = 1$
1  if $i = m + 1$ then
2      if $g$ satisfies $\mathcal{P}$ then
3          return $\{g\}, i_{\text{completed}}$
4      end
5      return $\emptyset, i$
6  end
7  $\bar{\Delta} \leftarrow$ Sort($(\Delta^{(i)})^g, \prec$)
8  forall $\delta \in \bar{\Delta}$ do
9      $\gamma_{i+1} \leftarrow \delta$
10     $\delta' \leftarrow \delta^{g^-}$
11     find $u_{\delta'} \in U^{(i)}$ with $\beta_i^{u_{\delta'}} = \delta'$
12     $n' \leftarrow$ extend $n$ by $\gamma_{i+1}$
13     $K', j \leftarrow$ Backtrack($n', u_{\delta'}g, i + 1$)
14     $K \leftarrow K \cup K'$
15     if $j < i$ then
16         return $\emptyset, j$
17     end
18 end
19 $i_{\text{completed}} \leftarrow \min\{i_{\text{completed}}, i\}$
20 return $K, i$
```
Algorithm 3.2: Backtrack subgroup search with elementary pruning

Algorithm 3.2 implements the improvements of this section and it is recursively referred to as `Backtrack`. We introduce a new global variable $i_{\text{completed}}$, whose value is shared among all recursive calls and is initialized with m, to save the level up to which $G^{[i]} \cap G(\mathcal{P})$

is fully known. We make use of this in line *3* where we instruct to backtrack to the last completed level. This multi-level backtracking due to Lemma 3.3 is then executed in line *16*. Of course, that we may do this depends on the order in which we visit the children of a node. Therefore we sort the orbit with respect to $\prec$ in line *7* and complete $G^{[i]}$ before visiting any element from $G \setminus G^{[i]}$.

3.1.3 Double coset minimality

Suppose that we know subgroups $K, L \leq G$ such that

$$g \in G(\mathcal{P}) \quad \Longleftrightarrow \quad KgL \subseteq G(\mathcal{P}). \tag{3.2}$$

Then we need to consider only one element of every double coset KgL that is somehow representative for the double coset. If we had some kind of order on G we could restrict our backtrack search to elements which are minimal in KgL. To this end we can use an ordering of Ω as introduced in the last section. This also induces an ordering of G by $g \prec h$, if and only if B^g is lexicographically $\prec$-smaller than B^h, i. e. there exists $1 \leq k \leq m$ with $\beta_i^g = \beta_i^h$ for $1 \leq i < k$ and $\beta_k^g \prec \beta_k^h$. However, determining double coset-minimality in general is NP-hard (cf. [Luk93]) so we usually settle for weaker criteria. In this section we look at some necessary conditions for double coset-minimality. If these conditions do not hold for some node of our search tree then we can prune this node because it cannot lead to minimal elements.

Before we go into details we assure ourselves that we usually have non-trivial subgroups K, L for which (3.2) holds and double coset minimality may be a useful concept. Of course (3.2) is always fulfilled for the trivial groups $K = L = \langle () \rangle$. When we search for a subgroup we do not need to distinguish between K and L, it is enough to consider the double coset KgK. We start with trivial K and add all elements of $G(\mathcal{P})$ that we find during the backtrack search to K and thus iteratively enlarge K. When performing a coset representative search [HEO05] suggests that it might be beneficial to solve one related subgroup problem first to obtain non-trivial K or L and prune the tree based on this knowledge. We will come back to this later in Section 3.1.5.

We now look at two necessary criteria for double coset minimality elements. These criteria help us to establish that a node in our search tree does not lead to double coset minimal elements and hence can be pruned.

Lemma 3.4. Let $n := (\gamma_1, \ldots, \gamma_j)$ be a node in the search tree. Suppose that $\beta_j \in \beta_i^{K_{(\beta_1,\ldots,\beta_{i-1})}}$ for some $i \leq j$. If elements $g \in \operatorname{coset}(n)$ are minimal in their coset KgL, then $\gamma_i \preceq \min \gamma_j^{L_{(\gamma_1,\ldots,\gamma_{i-1})}}$.

Proof. As $\beta_j \in \beta_i^{K_{(\beta_1,\ldots,\beta_{i-1})}}$ there exists $h_1 \in K_{(\beta_1,\ldots,\beta_{i-1})}$ with $\beta_j = \beta_i^{h_1}$. Suppose to the contrary that there exists a $\gamma \prec \gamma_i$ with $\gamma \in \gamma_j^{L_{(\gamma_1,\ldots,\gamma_{i-1})}}$. Then $\gamma = \gamma_j^{h_2}$ for some $h_2 \in L_{(\gamma_1,\ldots,\gamma_{i-1})}$, and for the product holds $h_1 g h_2 \in KgL$. We also have $h_1 g h_2 \prec g$ because $\beta_i^{h_1 g h_2} = \gamma \prec \gamma_i = \beta_i^g$ and $\beta_k^{h_1 g h_2} = \beta_k^g$ for $k < i$. This contradicts the minimality of g in KgL. □

Note that the precondition is always fulfilled for $i = j$, as clearly $\beta_j \in \beta_j^{K_{(\beta_1,\ldots,\beta_{j-1})}}$. To apply Lemma 3.4 in practice we need a base for L starting with $\gamma_1, \ldots, \gamma_{i-1}$, which may be expensive to compute. In an implementation we have to find a balance between the costs of a base change and the potential gain by pruning. One strategy could be to apply this lemma only in two situations: First, because of our ordering $\prec$ we have $(\beta_1, \dots, \beta_{i-1}) = (\gamma_1, \dots, \gamma_{i-1})$ up to some i for the first nodes in tree. For these we can readily apply the lemma since we already have a suitable base. Second, we could apply the lemma for tree nodes that are close to the root. By doing that, we limit the number of required base changes and can hope to prune large subtrees.

Lemma 3.5. Let $n := (\gamma_1, \dots, \gamma_i)$ be a node in the search tree and $s := |\beta_i^{K_{(\beta_1,\dots,\beta_{i-1})}}|$. If elements $g \in \operatorname{coset}(n)$ are minimal in their double coset KgL, then γ_i cannot be among the $s-1$ greatest (w.r.t. $\prec$) elements of its orbit under $G_{(\gamma_1,\dots,\gamma_{i-1})}$.

Proof. Consider the set $\Gamma := \{\beta_i^{hg} : h \in K_{(\beta_1,\dots,\beta_{i-1})}\}$, which has cardinality $|\Gamma| = s$ and includes $\gamma_i = \beta_i^g \in \Gamma$. All elements of Γ are in the same $G_{(\gamma_1,\dots,\gamma_{i-1})}$-orbit because for every $\gamma = \beta_i^{hg} \in \Gamma$ we have $\gamma^{g^- h^- g} = \gamma_i$ and $g^- h^- g \in G_{(\gamma_1,\dots,\gamma_{i-1})}$. Furthermore, for each $h \in K_{(\beta_1,\dots,\beta_{i-1})}$ it holds that $hg \in KgL$. So by minimality of g it follows that

$$g \preceq hg \quad \forall h \in K_{(\beta_1,\dots,\beta_{i-1})}. \tag{3.3}$$

Because we have $\beta_j^g = \gamma_j = \beta_j^{hg}$ for $1 \le j \le i-1$ and every $h \in K_{(\beta_1,\dots,\beta_{i-1})}$, equation (3.3) implies that $\beta_i^g = \gamma_i \preceq \beta_i^{hg}$ for every $h \in K_{(\beta_1,\dots,\beta_{i-1})}$, so $\gamma_i = \min \Gamma$. Thus the orbit $\gamma_i^{G_{(\gamma_1,\dots,\gamma_{i-1})}}$ must contain $|\Gamma| - 1 = s - 1$ elements after γ_i. □

We can apply Lemma 3.5 without a base change. When we are at a node $n = (\gamma_1, \dots, \gamma_{i-1})$ in the search tree, the extension for the children of n are given by all $\gamma_i \in (\Delta^{(i)})^g$ for a $g \in \operatorname{coset}(n)$. We can rewrite this set as

$$(\Delta^{(i)})^g = \beta_i^{G^{[i]}g} = \gamma_i^{g^- G_{(\beta_1,\dots,\beta_{i-1})} g} = \gamma_i^{G_{(\gamma_1,\dots,\gamma_{i-1})}}.$$

So by Lemma 3.5 we can ignore the $s-1$ greatest elements of $(\Delta^{(i)})^g$ when we visit child nodes. Because we have a strong generating set for K by construction of K (cf. Section 3.1.2) computing s is not too difficult. Algorithm 3.3 in the next section on page 32 provides an example with pruning based on Lemma 3.5.

3.1.4 Problem-dependent pruning

Besides these general pruning methods which are independent of our problem, we can also find simplifications of the search tree that come from $\mathcal{P}$. By definition of our search tree in Definition 3.1 on page 27, for a node $n := (\gamma_1, \dots, \gamma_{i-1})$ of our search tree we have all $(\gamma_1, \dots, \gamma_{i-1}, \gamma_i)$ as children with $\gamma_i \in (\Delta^{(i)})^g$ and a $g \in \operatorname{coset}(n)$. Some properties $\mathcal{P}$ may imply constraints on the base image. That means, we may find a set $\Omega_{\mathcal{P}}(n)$ dependent on the node n and the property $\mathcal{P}$ such that the following holds: If $\gamma_i \notin \Omega_{\mathcal{P}}(n)$ there cannot be a $g \in G(\mathcal{P}) \cap \operatorname{coset}(n)$ with $\beta_i^g = \gamma_i$. In other words, if $\gamma_i \notin \Omega_{\mathcal{P}}(n)$ we may skip the child γ_i. Note that we usually do not know anything about the reverse direction. The membership $\gamma_i \in \Omega_{\mathcal{P}}(n)$ has just to be a necessary condition for $\operatorname{coset}(n') \cap G(\mathcal{P}) \neq \emptyset$ where $n' := (\gamma_1, \dots, \gamma_{i-1}, \gamma_i)$. Furthermore, it may happen that we do not have such restrictions and thus $\Omega_{\mathcal{P}}(n) = \Omega$. We will see examples for non-trivial $\Omega_{\mathcal{P}}(n)$ shortly.

```
Input: BSGS for group $G$ with transversals $U^{(i)}$ and basic orbits $\Delta^{(i)}$, node $n$,
       corresponding $g \in \mathrm{coset}(n)$, level $i$, completed level $i_{\text{completed}}$ (global variable),
       known generators $K_0$ for a subgroup $\langle K_0 \rangle \leq G(\mathcal{P})$, level limit $i_{\text{limit}}$
Output: $K$ generating set of $\langle K \rangle = G(\mathcal{P})$ if $i = 1$
if $i = m + 1$ then                                        // visit leaves
    if $g$ satisfies $\mathcal{P}$ then
        return $\{g\}, i_{\text{completed}}$
    end
    return $\emptyset, i$
end
$K \leftarrow K_0$
$s_{\text{prune}} \leftarrow |(\Delta^{(i)})^g|$
$\bar{\Delta} \leftarrow \texttt{Sort}((\Delta^{(i)})^g, \prec)$
forall $\delta \in \bar{\Delta}$ do
    $K_{\text{stab}} \leftarrow \langle K \rangle_{(\beta_1, \ldots, \beta_{i-1})}$
    if $s_{\text{prune}} < |\beta_i^{K_{\text{stab}}}|$ then
        break
    end
    if $\delta \notin \Omega_{\mathcal{P}}(n)$ then                 // child restriction
        next
    end
    $\delta' \leftarrow \delta^{g^-}$
    find $u_{\delta'} \in U^{(i)}$ with $\beta_i^{u_{\delta'}} = \delta'$
    $\gamma_{i+1} \leftarrow \delta$
    $n' \leftarrow$ extend $n$ by $\gamma_{i+1}$
    $K', j \leftarrow \texttt{Backtrack}(n', u_{\delta'} g, i + 1, K, i_{\text{limit}})$
    $K \leftarrow K \cup K'$
    if $j < i$ then
        return $K, j$
    end
    $s_{\text{prune}} \leftarrow s_{\text{prune}} - 1$
end
$i_{\text{completed}} \leftarrow \min\{i_{\text{completed}}, i\}$
return $K, i$
```

Algorithm 3.3: Subgroup search with elementary double coset pruning

Algorithm 3.3 further extends Algorithm 3.2 with this $\mathcal{P}$-dependent pruning and pruning based on double coset minimality due to Lemma 3.5. We check for problem-dependent restrictions on line *15*. Lines *8* and *12* contain code for double coset minimality pruning. As already mentioned, K is by construction a strong generating set with respect to the base B of the input G. Hence we can easily compute the i-th element $K_{\text{stab}} := K^{[i]} = K_{(\beta_1,\dots,\beta_{i-1})}$ of the subgroup chain of K. This computation is performed in line *11*. Note that we speak of K in this text and formally use $\langle K \rangle$ in the algorithm because in the text we mean K to be whole known subgroup, whereas in the algorithm K is only a set of generators for $G(\mathcal{P})$. With the subgroup K_{stab} we then compute the i-th basic orbit $\beta_i^{K_{\text{stab}}}$ of K, whose length we need in our double coset pruning. Finally, s_{prune} is set to the number of elements in $\bar{\Delta}$. If this drops below $|\beta_i^{K^{[i]}}|$ in line *12* we can stop by Lemma 3.5.

In the following we will examine specializations for the set stabilization and group intersection problem. Both [Ser03, Sec. 9.1.2] and [HEO05, Sec. 4.6] contain problem-specific pruning methods for other problems like centralizers and normalizers.

Setwise stabilizer

Given $G \leq \mathrm{Sym}(\Omega)$ and a set $\Delta \subseteq \Omega$, we want to find generators of the setwise stabilizer G_Δ of Δ. We start with computing a BSGS (B, S) for G in which all elements of Δ precede $\Omega \setminus \Delta$ in the base. Let k be the first index for which $\beta_k \notin \Delta$. We can always find such an index by extending the base with possibly redundant base points from $\Omega \setminus \Delta$. Then we already have computed a subgroup $G^{[k]} = G_{(\beta_1,\dots,\beta_{k-1})} = G_{(\Delta)} \leq G_\Delta$, namely the pointwise stabilizer of Δ.

[Ser03, Sec. 9.1.2] also contains the observation that we only have to traverse the search tree up to level $k-1$. Note that for a node $n = (\gamma_1, \dots, \gamma_{k-1})$ of the search tree we either have $\mathrm{coset}(n) \subseteq G_\Delta$ (in other terms $\{\gamma_1, \dots, \gamma_{k-1}\} \subseteq \Delta$) or $\mathrm{coset}(n) \cap G_\Delta = \emptyset$. Thus, if $\mathrm{coset}(n) \subseteq G_\Delta$ holds, Lemma 3.3 on page 28 tells us that one generator from the coset is enough. Naturally, in the other case $\mathrm{coset}(n) \cap G_\Delta = \emptyset$ we also prune the node.

For the nodes at level $i < k-1$ we also do not have to consider all children. It is obviously enough to traverse all children where $\gamma_{i+1} \in \Delta$. So with the notation from above,

$$\Omega_{\mathcal{P}}(n) \equiv \Delta \quad \text{for } i < k-1. \tag{3.4}$$

Because we have found a lot of specializations for the set stabilizer problem we summarize all changes in a separate algorithm description. Algorithm 3.5 computes the setwise stabilizer G_Δ and calls Algorithm 3.4 as `Backtrack`. Algorithm 3.4 specializes Algorithm 3.3 by three features.

First, we limit the length of the base image we consider to i_{limit}, which is originally computed in line *2* of Algorithm 3.5. When in the backtrack search of Algorithm 3.4 this limit is reached at the check in line *1*, we abort the search. In the special case that we are in the first branch of the search tree, corresponding to g being the identity, we add the generators of $G^{[i]} = G_{(\Delta)}$, which is a subgroup of G_Δ, to our result set K.

Second, we apply the $\Omega_{\mathcal{P}}(n)$ restriction. The images of the first base points, which are by construction in Algorithm 3.5 a subset of Δ, have to remain in Δ as described in (3.4). This check is performed in line *20*.

Input: BSGS for group $G = \langle S \rangle$ with transversals $U^{(i)}$ and basic orbits $\Delta^{(i)}$, node n, corresponding $g \in \mathrm{coset}(n)$, level i, completed level $i_{\text{completed}}$ *(global variable)*, known generators K_0 for setwise stabilizer $\langle K_0 \rangle \leq G_\Delta$, level limit i_{limit}

Output: K generating set of $\langle K \rangle = G_\Delta$ if $i = 1$

```
if $i = i_{\text{limit}}$ then                                  // visit leaves
    if $\Delta^g = \Delta$ then
        if $g = ()$ then
            $K \leftarrow S \cap G^{[i]}$
        else
            $K \leftarrow \{g\}$
        end
        return $K, i_{\text{completed}}$
    end
    return $\emptyset, i$
end
$K \leftarrow K_0$
$s_{\text{prune}} \leftarrow |(\Delta^{(i)})^g|$
$\bar{\Delta} \leftarrow$ Sort$((\Delta^{(i)})^g, \prec)$
forall $\delta \in \bar{\Delta}$ do
    $K_{\text{stab}} \leftarrow \langle K \rangle_{(\beta_1,\ldots,\beta_{i-1})}$
    if $s_{\text{prune}} < |\beta_i^{K_{\text{stab}}}|$ then
        break
    end
    if $\delta \notin \Delta$ then                               // child restriction
        break
    end
    $\delta' \leftarrow \delta^{g^-}$
    find $u_{\delta'} \in U^{(i)}$ with $\beta_i^{u_{\delta'}} = \delta'$
    $\gamma_{i+1} \leftarrow \delta$
    $n' \leftarrow$ extend $n$ by $\gamma_{i+1}$
    $K', j \leftarrow$ Backtrack$(n', u_{\delta'} g, i+1, K, i_{\text{limit}})$
    $K \leftarrow K \cup K'$
    if $j < i$ then
        return $K, j$
    end
    $s_{\text{prune}} \leftarrow s_{\text{prune}} - 1$
end
$i_{\text{completed}} \leftarrow \min\{i_{\text{completed}}, i\}$
return $K, i$
```

Algorithm 3.4: Set stabilizer search with elementary double coset pruning

```
Input: group $G$, set $\Delta$
Output: $K$ generating set of $\langle K\rangle = G_\Delta$ setwise stabilizer of $\Delta$ in $G$
1 Compute a BSGS $B, S$ for $G$ that starts with $\Delta$
2 $k \leftarrow$ first index of in $B$ such that $\beta_k \notin \Delta$
3 $i_{\text{completed}} \leftarrow |B|$
4 $K \leftarrow \emptyset$
5 $K, j \leftarrow$ Backtrack$((), 1, 1, i_{\text{completed}}, K, k)$
6 return $K$
```

Algorithm 3.5: Set stabilizer search setup

Third, if the base point image $\delta = \gamma_{i+1}$ is not in Δ we can even abort the loop. All remaining elements in $\bar{\Delta}$ are $\prec$-greater than δ, which is not in Δ. By construction of $\prec$ and the ordering of our base, all elements $\prec$-greater than δ can neither be in Δ.

Group intersection

Given $G, H \leq \mathrm{Sym}(\Omega)$ with $|G| \leq |H|$, we are looking for generators of $G \cap H$. We compute an arbitrary base $B = (\beta_1, \dots, \beta_m)$ for G and a base for H which begins with B. Let $\Delta_H^{(i)}$ be the i-th fundamental orbit of H.

First of all we build up our search tree with respect to G. When we process a node $n := (\gamma_1, \dots, \gamma_{k-1})$ of the tree, we may compute the child node restriction as:

$$\Omega_{\mathcal{P}}(n) = (\beta_k^{H_{(\beta_1,\dots,\beta_{k-1})}})^h = (\Delta_H^{(k)})^h$$

where $h \in H$ is an element with $\beta_i^h = \gamma_i$ for $i \leq k-1$. We can find h by sifting through the first $k-1$ transversals of H. This restriction says in other words: If and only if $\gamma_k \in \Omega_{\mathcal{P}}(n)$ then there exists a $h \in H$ which has the same image of the first k base points. So especially $\gamma_k \in \Omega_{\mathcal{P}}(n)$ is a necessary condition for $\mathrm{coset}(n') \cap G(\mathcal{P}) = \mathrm{coset}(n') \cap H \neq \emptyset$ where $n' := (\gamma_1, \dots, \gamma_{k-1}, \gamma_k)$.

When our tree traversal reaches level m at node n_m we have to check by sifting whether the unique element $g \in \mathrm{coset}(n_m)$ is also a member of H because we so far have only ensured that there exists an $h \in H$ with the same image of the first m base points.

3.1.5 Coset representative search

When $G(\mathcal{P})$ is a coset and we look for a coset representative instead of a subgroup we can proceed as in the subgroup case that we have studied in detail before. The most important thing of course is that we can abort the search when we have found one element satisfying $\mathcal{P}$.

Additionally, if some non-trivial K or L are known beforehand, the search tree may be pruned efficiently by double coset minimality constraints. Especially if several coset representatives for different right-hand sides are sought, it might be worthwhile to solve the related left-hand side subgroup problem first. [HEO05] states that is generally helpful for larger groups to solve a subgroup problem before the coset problem to avoid "disasters" where the search gets stuck in the tree.

Applied to the set stabilizer case, where the coset case is by far more important than for group intersections, this means the following. Assume that we have sets $\Delta, \Gamma_1, \Gamma_2, \ldots, \Gamma_m \subseteq \Omega$ and want to know whether elements $g_1, \ldots, g_m \in G$ exist such that $\Delta^{g_1} = \Gamma_1, \ldots, \Delta^{g_m} = \Gamma_m$. Then it may be beneficial to compute G_Δ first because we can use it as K for double coset pruning as clearly $\Delta^g = \Gamma \iff \Delta^{(G_\Delta g)} = \Gamma$. Of course we also have to adapt the constraint (3.4) and replace it by $\Omega_{\mathcal{P}}(n) \equiv \Gamma$.

3.2 Partition backtracking

The backtracking approach we have seen so far has one big disadvantage: it works only with one (base) point at the same time. However, it is often possible to put the knowledge of all prior decisions to good use and further prune the tree. To illustrate this we take a look at an example from graph theory.

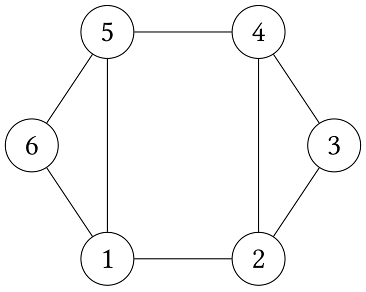


Figure 3.3: Graph automorphism example

Example 3.6. If we want to compute all automorphism of the graph depicted in Figure 3.3, i. e. all permutations that leave the graph structure unchanged, we also can employ backtracking. First we observe that all automorphisms must preserve the triangular structure of $\{1, 5, 6\}$ and $\{2, 3, 4\}$. More formally, we have an unordered partition $\Pi := \{\{1, 5, 6\}, \{2, 3, 4\}\}$ of all vertices which every automorphism must not change. Starting our backtrack search at vertex 1, we can choose among 1, 2, 4 and 5, which have the same valency, for the image of 1. Choosing, for instance, 2 as the image of 1 has immediate implications on the possibilities we have for the remaining vertices. Because we have to preserve Π it follows that $\{5, 6\}$ has to be mapped onto $\{3, 4\}$. Due to the degrees of these vertices we then must map 3 onto 6 and 4 onto 5. This gives us one automorphism $(1\,2)(3\,6)(4\,5)$ of the graph. Note that without the knowledge of the conservation of Π we might have tried to choose 4 as the image of 2 during our backtrack search and only later found out that it does not lead to a correct solution.

Backtracking methods based on partitions were introduced by McKay in the 1980s in the realm of graph automorphisms (cf. [McK81] and the software [`nauty`]). Later on in 1991 Leon successfully applied partition-based backtracking to group theoretical problems (cf. [Leo91, Leo97]), extending his previous work on automorphisms of combinatorial objects such as matrices and linear codes (cf. [Leo84]). Theißen worked separately at the difficult special case of computing normalizers (cf. [The97]). In this section we concentrate on the application of Leon's ideas to the set stabilizer and group intersection problem, which – as in the case of the classical backtrack search – are only a sample of problems where this method is applicable. The general framework is still abstract enough to allow the search for automorphisms of discrete and combinatorial objects such as matrices, graphs or linear codes (cf. [Leo91]). A patched derivative of Leon's original partition backtracking implementation, for instance, is still used in the GAP package [`GUAVA`] for computation with codes.

3.2.1 Introduction to partitions

First we need to clarify the central term of a partition that has already been introduced informally in our graph example. It will become clear immediately that we need ordered instead of unordered partitions to backtrack properly.

Definition 3.7. An **ordered partition** $\Pi = (\Pi_1, \dots, \Pi_k)$ of Ω is a sequence of non-empty, pairwise disjoint subsets $\Pi_i \subseteq \Omega$ such that $\bigcup_{i=1}^k \Pi_i = \Omega$. The sets Π_i are called **cells** of Π. We denote the **length** of Π by $|\Pi| := k$ and the set of all ordered partitions by $\mathrm{OP}(\Omega)$. The group $\mathrm{Sym}(\Omega)$ acts cellwise on ordered partitions: $\Pi^g := (\Pi_1^g, \dots, \Pi_k^g)$ for every $g \in \mathrm{Sym}(\Omega)$.

We prefer ordered partitions to unordered partitions because two ordered partitions $\Pi, \Sigma \in \mathrm{OP}(\Omega)$ with $|\Pi| = |\Sigma| = |\Omega|$, i. e. consisting of only single-element cells, induce exactly one permutation $g \in \mathrm{Sym}(\Omega)$ by $\Pi^g = \Sigma$. In the following we refer to ordered partitions of Ω simply as partitions. An important relation between partitions is that of a refinement:

Definition 3.8. Let $\Pi = (\Pi_1, \dots, \Pi_l)$ and $\Sigma = (\Sigma_1, \dots, \Sigma_m)$ be partitions. We say that Π is a **refinement** of Σ, $\Pi \leq \Sigma$, if the cells of Σ are a consecutive union of cells of Π. Formally, $\Sigma_i = \bigcup_{j=k_{i-1}+1}^{k_i} \Pi_j$ for some indices $0 = k_0 < k_1 < \cdots < k_m = l$. A refinement is **strict** if $|\Pi| > |\Sigma|$ and we write $\Pi \lneq \Sigma$ in this case.

The central concept of our partition backtracking will be a refinement process that in some way harmonizes with the property $\mathcal{P}$ we are looking for.

Definition 3.9. A $\mathcal{P}$**-refinement** $\mathcal{R}$ is a mapping $\mathcal{R} : \mathrm{OP}(\Omega) \to \mathrm{OP}(\Omega)$ such that

- $\mathcal{R}(\Pi) \leq \Pi$ for $\Pi \in \mathrm{OP}(\Omega)$ and,
- if $g \in G(\mathcal{P})$ and $\Pi \in \mathrm{OP}(\Omega)$ it holds that

$$\mathcal{R}(\Pi)^g = \mathcal{R}(\Pi^g). \tag{3.5}$$

In other words, the operation of a $\mathcal{P}$-refinement and every $g \in G(\mathcal{P})$ on partitions commute. This means, if g is unknown, we can potentially gain information on g because we know how it acts on a finer partition with less degrees of freedom. To actually create refinements of partitions we can use an intersection of partitions.

Definition 3.10. Let $\Pi = (\Pi_1, \dots, \Pi_l)$ and $\Sigma = (\Sigma_1, \dots, \Sigma_m)$ be partitions. We define the **intersection** $\Pi \wedge \Sigma$ as the partition with the non-empty sets $\Pi_i \cap \Sigma_j$ for $1 \leq i \leq l$, $1 \leq j \leq m$ as cells, ordered by the following rule: $\Pi_{i_1} \cap \Sigma_{j_1}$ precedes $\Pi_{i_2} \cap \Sigma_{j_2}$ if and only if $i_1 < i_2$ or $i_1 = i_2$ and $j_1 < j_2$.

Lemma 3.11. Let $\Pi, \Sigma \in \mathrm{OP}(\Omega)$. The intersection $\Pi \wedge \Sigma$ is a refinement of Π. The reverse statement does not hold.

Proof. By definition of the intersection we have $\Pi \wedge \Sigma = (\Pi_1 \cap \Sigma_1, \Pi_1 \cap \Sigma_2, \dots, \Pi_1 \cap \Sigma_m, \Pi_2 \cap \Sigma_1, \Pi_2 \cap \Sigma_2, \dots)$, which is a refinement of Π. The order in which we have defined the intersection to work with cells is also the reason why $\Pi \wedge \Sigma$ is not a refinement of Σ. □

Example 3.12. Consider the set $\Omega = \{1, 2, \dots, 7\}$. We write $\Pi := (1\,3\,5 \,|\, 2\,4 \,|\, 6\,7)$ in short for the partition $(\{1, 3, 5\}, \{2, 4\}, \{6, 7\})$. We have that $\Sigma := (1\,3 \,|\, 5 \,|\, 2\,4 \,|\, 6 \,|\, 7) \leq \Pi$ is a strict refinement of Π. Because the order into which cells are split up is not relevant, $\Sigma' := (5 \,|\, 1\,3 \,|\, 4 \,|\, 2 \,|\, 6\,7) \leq \Pi$ is another strict refinement of Π. We can "isolate" elements $\alpha \in \Omega$ of a partition by intersecting with $\mathrm{I}_\alpha := (\alpha \,|\, \Omega \setminus \{\alpha\})$. For example, to isolate 5 in Π we compute $\Pi \wedge \mathrm{I}_5 = (5 \,|\, 1\,3 \,|\, 2\,4 \,|\, 6\,7)$.

Before we step into technical details we have a brief look at the idea of partition backtracking. Let us assume we have $\Pi, \Sigma \in \mathrm{OP}(\Omega)$ such that for some (possibly unknown) $g \in G(\mathcal{P})$ the relation $\Pi^g = \Sigma$ holds. As mentioned before, $\Pi^g := (\Pi_1^g, \dots, \Pi_k^g)$ means cellwise action of g on the partition. For instance, the trivial partitions $\Pi = \Sigma = (\Omega)$ are an obvious starting point. The finer Π and Σ are, i. e. the more cells they have, the more information we get about $g \in G(\mathcal{P})$. If both Π and Σ are **discrete**, that means all cells consist of a single element, the permutation g is uniquely determined.

The way to get there are $\mathcal{P}$-refinements. We start with a pair $\hat{\Pi}, \hat{\Sigma}$ of ordered partitions such that $\hat{\Pi}^g = \hat{\Sigma}$ for some $g \in G(\mathcal{P})$. Then we try to find $\mathcal{P}$-refinements $\mathcal{R}$ that actually yield finer partitions $\Pi := \mathcal{R}(\hat{\Pi}) \lneq \hat{\Pi}$ and $\Sigma := \mathcal{R}(\hat{\Sigma}) \lneq \hat{\Sigma}$. For such a refinement $\mathcal{R}$ we obtain $\Pi^g = \mathcal{R}(\hat{\Pi})^g = \mathcal{R}(\hat{\Pi}^g) = \Sigma$ by the $\mathcal{P}$-refinement property (3.5). We can iterate this process until we cannot find a better, strict refinement. If the resulting Π and Σ are not discrete we resort to backtracking as follows: We pick one cell index $1 \leq j \leq |\Pi|$ with $|\Pi_j| \geq 2$ and one $\alpha \in \Pi_j$. Because $\Pi^g = \Sigma$ and especially $\Pi_j^g = \Sigma_j$ the image α^g of α has to be some $\beta \in \Sigma_j$. In a backtracking manner we probe each of this possible image candidates β.

Definition 3.13. A **backtrack refinement** $\mathcal{B}_\alpha$ is a function $\mathrm{OP}(\Omega) \to \mathrm{OP}(\Omega)$ defined as

$$\mathcal{B}_\alpha(\Pi) := \Pi \wedge (\{\alpha\}, \Omega \setminus \{\alpha\}).$$

So for the next iteration we set $\hat{\Pi} := \mathcal{B}_\alpha(\Pi)$ and $\hat{\Sigma} := \mathcal{B}_\beta(\Sigma)$ for some $\beta \in \Sigma_j$, assuming that there still is a $g \in G(\mathcal{P})$ with $\hat{\Pi}^g = \hat{\Sigma}$ (and $\alpha^g = \beta$). When we eventually reach a discrete partition Σ the pair Π, Σ defines a unique $g \in \mathrm{Sym}(\Omega)$ by $\Pi^g = \Sigma$. What is left to us is to check whether also $g \in G(\mathcal{P})$.

Example 3.14. For a small example of the idea we consider $G = S_3 = \langle (1\,2), (2\,3) \rangle$ and look for the stabilizer $G_{\{3\}}$ of 3 in G. We start with $\hat{\Pi} = \hat{\Sigma} = (\Omega) = (1\,2\,3)$. Because every $g \in G(\mathcal{P}) = G_{\{3\}}$ must map 3 onto itself a suitable strict $\mathcal{P}$-refinement $\mathcal{R}$ is $\mathcal{R} := I_3$ with I_α from Example 3.12. That means we isolate 3 into a single cell. We thus obtain $\Pi = I_3(\hat{\Pi}) = (3 \,|\, 1\,2)$ and $\Sigma = I_3(\hat{\Sigma}) = (3 \,|\, 1\,2)$. For every $g \in G(\mathcal{P})$ holds $\Pi^g = \Sigma$.

We cannot find another strict $\mathcal{P}$-refinement so we start backtracking. We have to pick the cell $1\,2$ of Π and choose $\mathcal{B}_1$ as backtrack refinement. Hence we have to consider the pairs $\mathcal{B}_1(\Pi), \mathcal{B}_1(\Sigma)$ and $\mathcal{B}_1(\Pi), \mathcal{B}_2(\Sigma)$ next. First we continue with $\hat{\Pi} := \mathcal{B}_1(\Pi) = (3\,|\,1\,|\,2)$ and $\hat{\Sigma} := \mathcal{B}_1(\Sigma) = (3\,|\,1\,|\,2)$. Both cells are discrete so we can read $g = ()$, which obviously is in $G(\mathcal{P})$. The second backtrack alternative is $\hat{\Pi} := \mathcal{B}_1(\Pi) = (3\,|\,1\,|\,2)$ and $\hat{\Sigma} := \mathcal{B}_2(\Sigma) = (3\,|\,2\,|\,1)$. We infer $g = (1\,2)$, which also is in $G(\mathcal{P})$. We have no more alternative left and we thus know $G(\mathcal{P})$ is complete with $G(\mathcal{P}) = \{(), (1\,2)\}$.

3.2.2 Search tree

We begin with a definition of an R-base that plays a role similar to the BSGS base in classical backtracking.

Definition 3.15. An **R-base** is a chain of refined partitions $\Pi^{(i)}$, $\hat{\Pi}^{(i)}$ together with a pair of $\mathcal{P}$- and backtrack refinement $\mathcal{R}^{(i)}$, $\mathcal{B}_{\alpha_i}$ such that

$$\hat{\Pi}^{(i)} = \mathcal{B}_{\alpha_i}(\Pi^{(i-1)}) \tag{3.6}$$

$$\Pi^{(i)} = \mathcal{R}^{(i)}(\hat{\Pi}^{(i)}) \tag{3.7}$$

holds for $1 \le i \le m$, starting with $\Pi_0 := \mathcal{R}^{(0)}(\Omega)$ where $\mathcal{R}^{(0)}$ is an initial $\mathcal{P}$-refinement. The chain ends at the first index m such that $\Pi^{(m)}$ is discrete.

We can ensure the finiteness of the chain by a careful choice of α_i for (3.6). As long as $\Pi^{(i)}$ is not discrete we find a cell of $\Pi^{(i)}$ that has at least two elements. Isolating one of them yields a strict refinement.

It is very important to note that an R-base depends not only on the group G to search in, but also on the property $\mathcal{P}$. Searches for different subgroups of the same group G will lead to different R-bases. So it takes the role of both base and child restriction $\Omega_{\mathcal{P}}(n)$ (cf. page 31) of classical backtracking.

Definition 3.16. Let $\Pi \in \mathrm{OP}(\Omega)$ be a partition that has been created from Ω by a series of refinements. We define $\operatorname{fix}(\Pi)$ as the ordered sequence of single-element cells of Π in the order in which they appeared in the refinement process.

Example 3.17. To clarify the meaning of fix we look at the following partitions and their "fix" points

$$\begin{array}{rlll}
 & \Pi^{(0)} = (1\,2\,3\,4\,5\,6\,7) & \qquad & \operatorname{fix}\Pi^{(0)} = \emptyset \\
\ge & \Pi^{(1)} = (1\,2\,3\,|\,4\,5\,6\,7) & & \operatorname{fix}\Pi^{(1)} = \emptyset \\
\ge & \Pi^{(2)} = (1\,|\,2\,3\,|\,4\,5\,6\,7) & & \operatorname{fix}\Pi^{(2)} = (1) \\
\ge & \Pi^{(3)} = (1\,|\,3\,|\,2\,|\,4\,5\,6\,7) & & \operatorname{fix}\Pi^{(3)} = (1,3,2) \\
\ge & \Pi^{(4)} = (1\,|\,3\,|\,2\,|\,4\,5\,|\,6\,7) & & \operatorname{fix}\Pi^{(4)} = (1,3,2) \\
\ge & \Pi^{(5)} = (1\,|\,3\,|\,2\,|\,4\,|\,5\,|\,6\,7) & & \operatorname{fix}\Pi^{(5)} = (1,3,2,4,5) \\
\ge & \Pi^{(6)} = (1\,|\,3\,|\,2\,|\,4\,|\,5\,|\,7\,|\,6) & & \operatorname{fix}\Pi^{(6)} = (1,3,2,4,5,7,6)
\end{array}$$

We are now ready to define our search tree for partition backtracking.

Definition 3.18. Let $\Pi^{(i)}$, $\hat{\Pi}^{(i)}$, $\mathcal{R}^{(i)}$, $\mathcal{B}_{\alpha_i}$ be an R-base. Our **search tree** is a tree with partitions $\Sigma^{(i)}$ as node labels and the following properties:

- The root at level 0 has the label $\Sigma^{(0)} := \Pi^{(0)}$.
- Let i be an intermediate level $0 \leq i < m$. Let b be the cell index of $\Pi^{(i)}$ which contains α_{i+1}. Then the node $\Sigma^{(i)}$ has the children

  $$\Sigma^{(i+1),\beta} := \mathcal{R}^{(i+1)}(\mathcal{B}_\beta(\Sigma^{(i)})) \quad \text{for every } \beta \in \Sigma_b^{(i)}$$

  for which there exists a $g \in G$ with $(\operatorname{fix} \Pi^{(i+1)})^g = \operatorname{fix} \Sigma^{(i+1),\beta}$. Here we use the superscript β in $\Sigma^{(i+1),\beta}$ only as a referencing index with no further computational meaning.

Definition 3.18 is a formal description of the general idea we discussed in the last section. We start with a partition $\Pi^{(0)}$ from our R-base and the root node $\Sigma^{(0)}$ which by definition fulfill the relationship

$$(\Pi^{(0)})^g = \Sigma^{(0)} \quad \forall g \in G(\mathcal{P}). \tag{3.8}$$

In the first level child nodes $\Sigma^{(1)}$ we first split up our search into all possible cases for the image of α_1. Naming $\hat{\Pi}^{(1)} := \mathcal{B}_{\alpha_1}(\Pi^{(0)})$ and $\hat{\Sigma}^{(1),\beta} := \mathcal{B}_\beta(\Sigma^{(0)})$, equation (3.8) implies that

$$(\hat{\Pi}^{(1)})^g = \hat{\Sigma}^{(1),\beta} \quad \forall g \in G(\mathcal{P}) \cap \{g \in G \ : \ \alpha_1^g = \beta\}. \tag{3.9}$$

We then apply the $\mathcal{P}$-refinement $\mathcal{R}^{(1)}$ and (3.9) implies together with the refinement property (3.5) that

$$(\Pi^{(1)})^g = \Sigma^{(1),\beta} \quad \forall g \in G(\mathcal{P}) \cap \{g \in G \ : \ \alpha_1^g = \beta\}. \tag{3.10}$$

Before we continue with the next backtrack round we would like to know if the restriction $\alpha_1^g = \beta$ leads to an unsatisfiable situation and we may thus skip this node. In general this is hard to decide and so we settle for weaker criteria: First of all the number of cells $|\Pi^{(1)}| = |\Sigma^{(1)}|$ has to be equal. Second, the sizes of corresponding cells $|\Pi_j^{(1)}| = |\Sigma_j^{(1)}|$ have to match for $1 \leq j \leq |\Pi^{(1)}|$. At last we check whether there exists at least a $g \in G$ which maps the single element cells $\operatorname{fix} \Pi^{(1)}$ onto $\operatorname{fix} \Sigma^{(1)}$. We could easily test this necessary requirement if we had a BSGS of G that starts with $\operatorname{fix} \Pi^{(1)}$.

Of course we would like to execute these checks on every level to prune the tree early. Because $\operatorname{fix} \Pi^{(m)}$ is known after the setup of the R-base and because $\operatorname{fix} \Pi^{(i)}$ is a prefix of $\operatorname{fix} \Pi^{(i+1)}$, we compute a BSGS for G with base $\operatorname{fix} \Pi^{(m)}$ once at the beginning. Using Algorithm 3.6 on this BSGS, we then can perform checks for $(\operatorname{fix} \Pi^{(i)})^g = \operatorname{fix} \Sigma^{(i),\beta}$ on every level i.

The leaves of the search tree, discrete partitions $\Sigma^{(m)}$, correspond uniquely to group elements g which may or may not be a member of $G(\mathcal{P})$. So the search tree may contain more than $|G(\mathcal{P})|$ leaves. By construction we only ensure that every $g \in G(\mathcal{P})$ corresponds to a leaf of the tree because we build up a chain of necessary conditions for membership by the $\mathcal{P}$-refinement property (3.5) on page 37.

Lemma 3.19. For every $g \in G(\mathcal{P})$ there exists a leaf $\Sigma^{(m)}$ of the search tree with $(\Pi^{(m)})^g = \Sigma^{(m)}$.

```
Input: $B = (\beta_1, \ldots, \beta_m)$, $S$ BSGS with transversals $U^{(i)}$, sequence of points
       $(\alpha_1, \ldots, \alpha_k)$
Output: $g \in G$ with $\beta_i^g = \alpha_i$ for $1 \leq i \leq k$, if such an element exists
$g \leftarrow ()$
for $i = 1$ to $k$ do
    $\delta \leftarrow \alpha_i^{g^-}$
    find $u_\delta \in U^{(i)}$ with $\beta_i^{u_\delta} = \delta$
    if no $u_\delta$ found then
        return $\emptyset$
    end
    $g \leftarrow u_\delta g$
end
return g
```

Algorithm 3.6: Recover group element from given (partial) base image

Proof. We proof this lemma by induction on the level i. Our induction hypothesis is that for every $g \in G(\mathcal{P})$ there exists a node $\Sigma^{(i)}$ at level i in the search tree with

$$(\Pi^{(i)})^g = \Sigma^{(i)}. \tag{3.11}$$

For $i = 0$ we have already seen this in (3.8).

So let the induction hypothesis be true up to level $i-1 < m$ and let $g \in G(\mathcal{P})$ arbitrary. Let further $\beta_i := \alpha_i^g$ and $\Sigma^{(i-1)}$ be a node with $(\Pi^{(i-1)})^g = \Sigma^{(i-1)}$. We have to show that we can always choose a child $\Sigma^{(i)}$ of $\Sigma^{(i-1)}$ with $(\Pi^{(i)})^g = \Sigma^{(i)}$. To see this let j be the cell index of $\Pi^{(i-1)}$ containing α_i. Because of (3.11) the cell $\Sigma_j^{(i-1)}$ must contain $\beta_i = \alpha_i^g$. Cell j is the cell we choose the backtrack refinement $\mathcal{B}_\beta$ from, so we can choose the child $\Sigma^{(i),\beta_i}$. Furthermore, by definition of β_i it holds that $(\mathcal{B}_{\alpha_i}(\Pi^{(i-1)}))^g = \mathcal{B}_{\beta_i}(\Sigma^{(i-1)})$. Thus after a $\mathcal{P}$-refinement we obtain $(\Pi^{(i)})^g = \Sigma^{(i),\beta_i} =: \Sigma^{(i)}$. □

Corollary 3.20. If $G(\mathcal{P})$ is a subgroup then $A := (\alpha_1, \ldots, \alpha_m)$ forms a base of $G(\mathcal{P})$.

Proof. Lemma 3.19 shows that every $g \in G(\mathcal{P})$ corresponds to a leaf $\Sigma^{(m)}$ of the search tree. Two different elements $g, h \in G(\mathcal{P})$, $g \neq h$ must correspond to two different leaves because otherwise they would have the same image of all elements of Ω, since leaves are discrete partitions. If g and h correspond to different leaves they must differ at least in one of the α_i images. Hence every $g \in G(\mathcal{P})$ is uniquely determined by its image of $(\alpha_1, \ldots, \alpha_m)$, so A is a base for $G(\mathcal{P})$. □

Remark 3.21. Note how we can emulate also a classical backtrack search. We set all $\mathcal{R}^{(i)}$ to the identity function and perform no $\mathcal{P}$-refinement at all. For the backtracking refinements we choose $\alpha_i := \beta_i$ where $(\beta_1, \ldots, \beta_m)$ is a base for G. In this way we iteratively test all partial base images of $(\beta_1, \ldots, \beta_j)$, $1 \leq j \leq m$, which is what classical backtracking does.

Algorithm 3.7, recursively referred to as `PartitionBacktrack`, depicts the search algorithm discussed so far. Starting with $i = 0$, we perform a depth-first search on the search tree defined by Definition 3.18. Lemma 3.19 guarantees that we hit every $g \in G(\mathcal{P})$. We prune the tree only very moderately, in Section 3.2.4 we will discuss more powerful methods.

Input: $\Pi^{(i)}$, $\hat{\Pi}^{(i)}$, $\mathcal{R}^{(i)}$, $\mathcal{B}_{\alpha_i}$ R-base for a property $\mathcal{P}$, partition $\Sigma^{(i)}$, level i, known generators K_0 for a subgroup $\langle K_0 \rangle \leq G(\mathcal{P})$
Output: K generating set of $\langle K \rangle = G(\mathcal{P})$ subgroup of G with property $\mathcal{P}$ if $i = 0$

```
if $i = m$ then
    if $g$ satisfies $\mathcal{P}$ then
        return $\{g\}$
    else
        return $\emptyset$
    end
end
$K \leftarrow K_0$
$j \leftarrow$ cell index of $\alpha_{i+1}$ in $\Pi^{(i)}$
forall $\beta \in \Sigma^{(i)}_j$ do
    $\Sigma^{(i+1)} \leftarrow \mathcal{R}^{(i+1)}(\mathcal{B}_\beta(\Sigma^{(i)}))$
    if $|\Sigma^{(i+1)}| \neq |\Pi^{(i+1)}|$ then
        next $\beta$
    end
    for $k = 1$ to $|\Sigma^{(i+1)}|$ do
        if $|\Sigma^{(i+1)}_k| \neq |\Pi^{(i+1)}_k|$ then
            next $\beta$
        end
    end
    find $g \in G$ with $(\mathrm{fix}\,\Pi^{(i+1)})^g = \mathrm{fix}\,\Sigma^{(i+1)}$
    if no such $g$ exists then
        next $\beta$
    end
    $K \leftarrow K \cup$ PartitionBacktrack($\Sigma^{(i+1)}, K, i+1$)
end
return $K$
```

Algorithm 3.7: Partition backtrack subgroup search with very basic pruning

3.2.3 Constructing an R-base

Now that we know what we need an R-base for we discuss the matter of how to construct one for our problem $\mathcal{P}$. This consists of two independent tasks, which we have to perform repeatedly: finding proper $\mathcal{P}$-refinements and choosing the right α_i for the backtrack refinements.

We begin with the choice of the α_i. Looking again at the search tree in Definition 3.18, we see that the size of the backtrack search is bounded by the product $d := |\Pi^{(0)}_{j_0}| \cdot |\Pi^{(1)}_{j_1}| \cdot \dots \cdot |\Pi^{(m)}_{j_m}|$ where j_i denotes the cell index of α_i in $\Pi^{(i)}$. Thus an obvious goal for the choice of α_i is to minimize the product d. A heuristic approach that also is easy to compute is to successively choose j_i such that $|\Pi^{(i)}_{j_i}|$ is minimal. This of course might not minimize the total product d. Nevertheless, [Leo97, p. 140] states that according to his experience this simple heuristic works as good as any other which is easy to implement. A computationally more expensive alternative would be to try multiple values of j_i for every i and choose the combination which yields minimal d. We postpone the discussion of this approach until the end of this section and turn to a completely different optimization: As we will see below, the choice of α_i may require a base change to allow for efficient R-base construction. To reduce the number of base changes, especially profitable for small groups, we could also try to choose α_i so that no base change is required as long as it does not increase d to much. In an implementation we have to find a compromise between the two α_i-selection strategies. We now turn to the second task in R-base construction, finding proper $\mathcal{P}$-refinements.

For every problem $\mathcal{P}$ we can choose from a pool of applicable $\mathcal{P}$-refinements, depending on $\mathcal{P}$. To avoid redundancies we are interested in finding $\mathcal{P}$-refinements that yield strict refinements of the input partition. In the following we examine $\mathcal{P}$-refinements which we can try to apply for stabilizing properties or group membership properties.

If $\mathcal{P}$ contains the stabilization of a set, we can choose the following:

Lemma 3.22. Let $\Delta \subseteq \Omega$ and $G(\mathcal{P}) \subseteq G_\Delta$. Then

$$\mathcal{R}_{\text{stab},\,\Delta}(\Pi) := \Pi \wedge (\Delta \mid \Omega \setminus \Delta) \tag{3.12}$$

is a $\mathcal{P}$-refinement.

Proof. Set $\Sigma := (\Delta \mid \Omega \setminus \Delta)$. We have to show that $(\Pi_i \cap \Sigma_j)^g = \Pi_i^g \cap \Sigma_j$ for all cell indices i, j and $g \in G(\mathcal{P})$. This obviously holds because by definition Σ is invariant under action of G_Δ and thus also of $G(\mathcal{P})$, yielding $(\Pi_i \cap \Sigma_j)^g = \Pi_i^g \cap \Sigma_j^g = \Pi_i^g \cap \Sigma_j$. □

Important to note is that we can apply this $\mathcal{P}$-refinement for a given Δ only once to obtain a strict refinement. The second application of $\mathcal{R}_{\text{stab},\,\Delta}$ has no more refinement power because Δ and $\Omega \setminus \Delta$ have already been separated.

Lemma 3.22 is just a reformulation of our observations from Section 3.1.4. The real power of refinements for groups lies in exploiting the orbit structures.

Definition 3.23. Let $G \leq \mathrm{Sym}(\Omega)$. We define $\Theta(G)$ as the ordered partition of Ω which consists of the orbits of Ω under action of G as cells, ordered by the minimal element of each orbit.

Example 3.24. Consider $G = \langle(1\,3)(2\,4)(5\,6), (1\,3)(2\,5)(4\,6)\rangle$. Then G has the distinct orbits $1^G = \{1, 3\}$ and $2^G = \{2, 4, 5, 6\}$. Because $\min 1^G = 1 < 2 = \min 2^G$ we obtain $\Theta(G) = (1\,3\,|\,2\,4\,5\,6)$.

Lemma 3.25. Let $H \leq \mathrm{Sym}(\Omega)$ and $G(\mathcal{P}) \subseteq H$. Let $\Pi \in \mathrm{OP}(\Omega)$ be given. Then

$$\mathcal{R}_{\text{orbit}, H, \Pi}(\Sigma) := \begin{cases} \Sigma \wedge \Theta(H_{\text{fix}\,\Pi})^t & \text{if there exists a } t \in H \text{ with } (\text{fix}\,\Pi)^t = \text{fix}\,\Sigma \\ \Sigma & \text{otherwise} \end{cases} \tag{3.13}$$

is a $\mathcal{P}$-refinement.

Proof. If no such t exists and the second case in (3.13) applies we have nothing to show. So let $g \in G(\mathcal{P})$ be arbitrary and $\Theta := \Theta(H_{\text{fix}\,\Pi})$. We assume there exists a $t \in H$ with $(\text{fix}\,\Pi)^t = \text{fix}\,\Sigma$. Let $u \in H$ with $(\text{fix}\,\Pi)^u = \text{fix}(\Sigma^g)$. Such a u exists because $u = tg$ would be a valid choice. To proof the lemma we have to show that $(\Sigma_i \cap \Theta_j^t)^g = \Sigma_i^g \cap \Theta_j^u$ for all cell indices i, j and $g \in G(\mathcal{P})$. We observe that $(\text{fix}\,\Pi)^{tgu^-} = \text{fix}\,\Pi$, so $tgu^- \in H_{\text{fix}\,\Pi}$ and thus $\Theta_j^{tgu^-} = \Theta_j$ or in other terms: $\Theta_j^{tg} = \Theta_j^u$. □

A first question when dealing with the $\mathcal{P}$-refinement of Lemma 3.25 is which partition Π to choose. Leon showed in [Leo91, Lemma 8] that, if there is a partition Π that yields a strict refinement $\mathcal{R}_{\text{orbit}, H, \Pi}(\Sigma)$ of Σ, we can always choose $\Pi := \Sigma$. When we search for elements of $G(\mathcal{P})$ and require membership in G it is thus sufficient to check whether $\mathcal{R}_{\text{orbit}, G, \Pi}(\Pi)$ yields a strict refinement of our previous partition Π. If we are looking for a group intersection $G \cap H$, we can try both $\mathcal{R}_{\text{orbit}, G, \Pi}$ and $\mathcal{R}_{\text{orbit}, H, \Pi}$.

The key for computing $\mathcal{R}_{\text{orbit}, G, \Pi}(\Pi)$ efficiently in practice is having a base for G that begins with fix Π, making generators of $G_{\text{fix}\,\Pi}$ available. So whenever fix Π changes during the R-base construction we have to change the base of all involved groups such that these start again with fix Π.

The two $\mathcal{P}$-refinements $\mathcal{R}_{\text{orbit}}$ and $\mathcal{R}_{\text{stab}}$ provide only a sample for the two subgroup problems we focus on. The interested reader may find in [Leo97, Fig. 2] more $\mathcal{P}$-refinements for other problems such as centralizers or automorphisms of combinatorial objects.

As an example Algorithm 3.8 shows the case of R-base construction for a set stabilization problem. We apply $R_{\text{stab}, \Delta}$ only once at the beginning because multiple applications are redundant as discussed above. It remains to try $\mathcal{R}_{\text{orbit}, G, \Pi^{(i)}}$ before resorting to backtracking. For the backtracking parameters α_i we choose the simple strategy of selecting a cell with minimal cardinality. R-base constructions for different problems differ in the choice of the considered refinements. As mentioned above, for $G \cap H$ we would include $\mathcal{R}_{\text{orbit}, H, \Pi^{(i)}}$ parallel to $\mathcal{R}_{\text{orbit}, G, \Pi^{(i)}}$.

Because we compute an R-base only once during the whole partition backtracking process it may be worthwhile to put more effort into the R-base construction in order to reduce the search tree size. The parameters we can adjust are: when to apply which refinement and the choice of the α_i. We could even enumerate all or randomly generate and test some refinement combinations to minimize the search tree size $d := |\Pi_{j_0}^{(0)}| \cdot |\Pi_{j_1}^{(1)}| \cdot \cdots \cdot |\Pi_{j_m}^{(m)}|$. However, only larger problems might benefit from such optimizations because the preprocessing effort gets too large relatively to the problem size.

```
Input: group $G \leq \mathrm{Sym}(\Omega)$ and set $\Delta \subseteq \Omega$
Output: R-base for $G(\mathcal{P}) = G_\Delta$
$\hat{\Pi}^{(0)} \leftarrow (\Omega)$
$\mathcal{R}^{(0)} \leftarrow \mathcal{R}_{\text{stab, }\Delta}$
$\Pi^{(0)} \leftarrow \mathcal{R}^{(0)}(\hat{\Pi}^{(0)})$
$i \leftarrow 0$
while $|\Pi^{(i)}| \neq |\Omega|$ do
    while $\mathcal{R}_{\text{orbit, }G,\,\Pi^{(i)}}(\Pi^{(i)}) \lneq \Pi^{(i)}$ do
        $\mathcal{R}^{(i)} \leftarrow \mathcal{R}^{(i)} \circ \mathcal{R}_{\text{orbit, }G,\,\Pi^{(i)}}$
        $\Pi^{(i)} \leftarrow \mathcal{R}_{\text{orbit, }G,\,\Pi^{(i)}}(\Pi^{(i)})$
        change base of $G$ such that it starts with $\mathrm{fix}\,\Pi^{(i)}$
    end
    if $|\Pi^{(i)}| \neq |\Omega|$ then
        $j \leftarrow$ cell index of $\Pi^{(i)}$ with minimal cardinality $|\Pi^{(i)}_j| \geq 2$
        $\alpha_i \leftarrow$ arbitrary element of $\Pi^{(i)}_j$
        $\hat{\Pi}^{(i+1)} \leftarrow \mathcal{B}_{\alpha_i}(\Pi^{(i)})$
        $\Pi^{(i+1)} \leftarrow \hat{\Pi}^{(i+1)}$
        change base of $G$ such that it starts with $\mathrm{fix}\,\Pi^{(i+1)}$
        $\mathcal{R}^{(i+1)} \leftarrow 1$ ;                       // identity refinement
    end
    $i \leftarrow i + 1$
end
```

Algorithm 3.8: R-base construction for a set stabilization problem

Still, the value of d is not the only factor determining the time needed for backtrack search. During the search we may have opportunities to prune the tree, which we cannot foresee in the construction phase. Hence there may be situations in which an R-base that is not optimal with respect to the product d offers a better search order for pruning.

3.2.4 Pruning the tree

We can use the same pruning techniques as for classical backtracking in Section 3.1.2, 3.1.3 and partly 3.1.4. We can gain the additional advantage from Lemma 3.3 and order the child nodes in our search tree. An appropriate order is sorting according to $\mathrm{fix}\,\Pi^{(m)}$, which contains a base $A := (\alpha_1, \ldots, \alpha_m)$ for $G(\mathcal{P})$ in the subgroup case as we have seen in Corollary 3.20 on page 41. This ordering ensures that we always work at $G^{[i]} \cap G(\mathcal{P})$ before $G \setminus G^{[i]}$. Thus we can prune nodes based on Lemma 3.3 and obtain automatically a strong generating set for $G(\mathcal{P})$ relative to the base A.

When we search for a setwise stabilizer G_Δ we can apply similar techniques as in Section 3.1.4 to prune the tree. During the R-base construction we choose the points α_i so that α_i lies in Δ, if possible. If for some index k we cannot choose $\alpha_k \in \Delta$ any longer we finish the construction as usual until a discrete partition is reached and we thus get a base $A = (\alpha_1, \ldots, \alpha_m)$ for G_Δ which starts with elements from Δ. In the partition backtrack search we then can abort our search at level k and add $G_{(\alpha_1,\ldots,\alpha_{k-1})} \subseteq G_\Delta$ to K, like in the classical backtracking case.

Furthermore, given $K \subseteq G(\mathcal{P})$, it still holds that when we add elements g to the result set K in Algorithm 3.7 we can restrict our search to elements minimal in their double coset KgK. So after the aforementioned ordering we can skip the last $s := |\alpha_i^{K_{(\alpha_1,\ldots,\alpha_{i-1})}}| - 1$ backtracking alternatives due to Lemma 3.5. We may also prune nodes based on Lemma 3.4, replacing $\beta_1, \ldots, \beta_j$ by $\alpha_1, \ldots \alpha_j$ as subgroup base.

[Leo97, Prop. 6] contains one more necessary condition for double coset minimality based on partitions. However, it is more expensive to check and Leon is unsure about its usefulness.

```
Input: $\Pi^{(i)}$, $\hat{\Pi}^{(i)}$, $\mathcal{R}^{(i)}$, $\mathcal{B}_{\alpha_i}$ R-base for a property $\mathcal{P}$, partition $\Sigma^{(i)}$, level $i$, completed
       level $i_{\text{completed}}$ (global variable), known generators $K_0$ for a subgroup
       $\langle K_0 \rangle \leq G(\mathcal{P})$
Output: $K$ generating set of $\langle K \rangle = G(\mathcal{P})$ subgroup of $G$ with property $\mathcal{P}$ if $i = 0$
if $i = m$ then
    if $g$ satisfies $\mathcal{P}$ then return $\{g\}, i_{\text{completed}}$;
    else return $\emptyset, i$;
end
$K \leftarrow K_0$;
$j \leftarrow$ cell index of $\alpha_{i+1}$ in $\Pi^{(i)}$;
$s_{\text{prune}} \leftarrow |\Sigma_j^{(i)}|$;
forall $\beta \in \texttt{Sort}(\Sigma_j^{(i)}, \prec)$ do
    $K_{\text{stab}} \leftarrow \langle K \rangle_{(\alpha_1,\ldots,\alpha_i)}$;
    if $s_{\text{prune}} < |\alpha_{i+1}^{K_{\text{stab}}}|$ then break;
    $\Sigma^{(i+1)} \leftarrow \mathcal{R}^{(i+1)}(\mathcal{B}_\beta(\Sigma^{(i)}))$;
    if $|\Sigma^{(i+1)}| \neq |\Pi^{(i+1)}|$ then next $\beta$;
    for $k = 1$ to $|\Sigma^{(i+1)}|$ do
        if $|\Sigma_k^{(i+1)}| \neq |\Pi_k^{(i+1)}|$ then
            next $\beta$;
        end
    end
    find $g \in G$ with $(\text{fix}\,\Pi^{(i+1)})^g = \text{fix}\,\Sigma^{(i+1)}$;
    if no such $g$ exists then
        next $\beta$;
    end
    $K', l \leftarrow \texttt{PartitionBacktrack}(\Sigma^{(i+1)}, K, i+1)$;
    $K \leftarrow K \cup K'$;
    if $l < i$ then return $K, l$;
    $s_{\text{prune}} \leftarrow s_{\text{prune}} - 1$;
end
$i_{\text{completed}} \leftarrow \min\{i_{\text{completed}}, i\}$;
return $K, i$;
```

Algorithm 3.9: Partition backtrack subgroup search with elementary pruning

Algorithm 3.9 extends Algorithm 3.7 from page 42 and improves its pruning capabilities. In line *8* we sort the backtracking refinements according to $\prec$. We define this ordering $\prec$ as $\alpha \prec \beta$ for $\alpha, \beta \in \Omega$ if and only if α precedes β in $\operatorname{fix}\Pi^{(m)}$, the order in which fix points appear during R-base construction. This ordering allows multi-level backtracking in line *24* according to Lemma 3.3. Additionally, we can check for double coset minimality in line *10* due to Lemma 3.5. So this Algorithm 3.9 incorporates the same pruning facilities as the classical backtracking Algorithm 3.3 on page 32.

3.2.5 Coset representative search

For partition backtracking the coset case differs more from the subgroup case than for classical backtracking. Naturally we can abort the search when we have found one element satisfying $\mathcal{P}$. But we also have to extend our search tree formulation and $\mathcal{P}$-refinement Definition 3.9 on page 37.

Definition 3.26. A **$\mathcal{P}$-refinement** is a pair of mappings $\mathcal{R}_L, \mathcal{R}_R : \mathrm{OP}(\Omega) \to \mathrm{OP}(\Omega)$ such that

- $\mathcal{R}_L(\Pi) \leq \Pi$ and $\mathcal{R}_R(\Pi) \leq \Pi$ for $\Pi \in \mathrm{OP}(\Omega)$ and,
- if $g \in G(\mathcal{P})$ and $\Pi, \Sigma \in \mathrm{OP}(\Omega)$ it holds that

$$\Pi^g = \Sigma \quad \Longrightarrow \quad \mathcal{R}_L(\Pi)^g = \mathcal{R}_R(\Sigma) \tag{3.14}$$

This definition contains the original one as special case where left and right refinement $\mathcal{R}_L = \mathcal{R}_R$ are equal. In the R-base construction we replace $\mathcal{R}$ by $\mathcal{R}_L$, in the search tree we use $\mathcal{R}_R$ instead of $\mathcal{R}$. The same arguments we used in Section 3.2.2 hold with minimal changes regarding (3.14) for this case. These show that a partition backtrack search on the extended R-base and search tree finds all elements of $G(\mathcal{P})$.

We can extend both $\mathcal{R}_{\text{stab}}$ and $\mathcal{R}_{\text{orbit}}$ refinement of Lemma 3.22 and 3.25 according to the new Definition 3.26. However, if we concentrate on the important case of set stabilizer cosets we do not have to change $\mathcal{R}_{\text{orbit}}$ because the precondition $G(\mathcal{P}) \subseteq H$ is still valid when we look for cosets of H, so we can compute with $\mathcal{R}_{\text{orbit}} = \mathcal{R}_{\text{orbit},L} = \mathcal{R}_{\text{orbit},R}$. We extend $\mathcal{R}_{\text{stab}}$ as follows:

Lemma 3.27. Let $\Delta, \Gamma \subseteq \Omega$ and $\Delta^g = \Gamma$ for all $g \in G(\mathcal{P})$. Then

$$\mathcal{R}_{\text{stab2, }\Delta,\text{ }\Gamma,L}(\Pi) := \Pi \wedge (\Delta \mid \Omega \setminus \Delta)$$
$$\mathcal{R}_{\text{stab2, }\Delta,\text{ }\Gamma,R}(\Sigma) := \Sigma \wedge (\Gamma \mid \Omega \setminus \Gamma)$$

is a $\mathcal{P}$-refinement.

Proof. Set $\Pi' := (\Delta \mid \Omega \setminus \Delta)$ and $\Sigma' := (\Gamma \mid \Omega \setminus \Gamma)$. We have to show that for every $g \in G(\mathcal{P})$ the following implication holds:

$$\Pi^g = \Sigma \;\Rightarrow\; (\Pi_i \cap \Pi'_j)^g = \Sigma_i \cap \Sigma'_j \quad \text{for all } i, j. \tag{3.15}$$

The condition $\Pi^g = \Sigma$ implies that $\Pi_i^g = \Sigma_i$ for all i. Because $g \in G(\mathcal{P})$ it holds that $\Delta^g = \Gamma$. Hence we have $(\Pi'_j)^g = \Sigma'_j$ and thus (3.15) holds. □

Problem	$\mathcal{P}$-refinements to use	
Setwise stabilizer	$\mathcal{R}_{\text{stab}}$, $\mathcal{R}_{\text{orbit}}$	Lemma 3.22,3.25
Group intersection	$\mathcal{R}_{\text{orbit}}$	Lemma 3.25
Set image	$\mathcal{R}_{\text{stab2}}$, $\mathcal{R}_{\text{orbit}}$	Lemma 3.27,3.25
Coset intersection		cf. [Leo97, Fig. 2]

Figure 3.4: Overview of $\mathcal{P}$-refinements for different problems

The table in Figure 3.4 summarizes the usage of all $\mathcal{P}$-refinements we examined in this chapter. The interested reader can find in [Leo97, Fig. 2] an extension of $\mathcal{R}_{\text{orbit}}$ that can be used in the coset case of group intersection, which is coset intersection. In this figure by Leon there are also $\mathcal{P}$-refinements described for various other problems such as matrix and group isomorphism.

4 Implementation

In this section we will take a closer look at implementation details and the author's implementation PermLib. The author has developed PermLib with three design goals in mind: performance, maintainability and ease of use. The only software dependency is the renowned Boost [Boost] C++ library. It is mostly used for smart pointers, which help to avoid memory leaks. Moreover, unit tests for low-level data structures are provided within the Boost test library to control the number of fundamental implementation flaws. More information about PermLib can be found in Appendix A.

In this chapter we first look at data structures for fundamental objects, which we have not discussed yet. Permutations are core elements of all algorithms, the same is true for partitions in partition-based backtracking. Because the performance of all other algorithms thus heavily depends on these basics we have to find data structures that allow us to perform all necessary operations fast and efficiently.

After a quick overview of the implementation we look at a series of experiments with PermLib. It will soon become clear that exhaustive tests go beyond the scope of this thesis, so we just scratch the surface of an extensive survey on computational group theory. In fact we use the experiments to gain some additional insights into the zoo of data structures we have explored so far. Furthermore, we compare the performance of PermLibwith the other public available state-of-the-art code of GAP and partly Magma. The former is at the disadvantage of being interpreted at runtime, so that a comparison of actual algorithm implementations beyond running time is difficult. The latter is not open source and was available only a special workstation to the author, so not for all experiments Magma results are available.

4.1 Low-level data structures

4.1.1 Permutation

Permutations are a central element for all algorithms we have discussed so far, so data structures for permutations play a key role for fast implementations. A trivial approach is to store a permutation $g \in \mathrm{Sym}(\Omega)$, $|\Omega| =: n$, as an n-dimensional array whose α-th cell contains the image α^g. In this way we have access to the image of an element α under action of g in $O(1)$ time and we can compute the inverse g^- in $\Omega(n)$ time. However, multiplying two permutations takes $\Omega(n^2)$ time. We call this form of permutation implementation **elementary permutations**.

A more sophisticated approach are so called **permutation words**, as for example described in [HEO05, Sec. 4.4.3]. The idea is to store g as a list (or word) of other permutations $(s_1 s_2 \dots s_l)$, representing the product $s_1 s_2 \cdots s_l$. When we choose the s_i from a fixed "representative" label set $S \leq \mathrm{Sym}(\Omega)$, which we store as elementary permutations, it is enough to store $g = s_1 s_2 \cdots s_l$ as pointers to elements s_i. Storing all permutations in double-linked lists allows multiplication in constant time $O(1)$ by list concatenation. Computing the image $\alpha^g = \alpha^{s_1 s_2 \cdots s_l}$, however, takes $\Omega(l)$ time. If we store for every $s \in S$ also $s^- \in S$ then we can invert g by reversing the list and replacing each pointer s_i by s_i^-, also summing up to $\Omega(l)$ time.

A canonical choice for the label set S is a strong generating set for a group G with respect to a certain base that we are interested in, adjoined by generator inverses. During computations, e. g. a base change, new generators may arise, first as words in the original generators. We can multiply them out completely and add them to the label set together with their inverse as elementary permutations.

Choosing between storing permutations completely as image arrays and as permutation words is a trade-off between speed of image access and multiplication. With increasing n an $\Omega(n^2)$ effort for frequently occurring multiplications becomes more and more infeasible. On the other hand, we often require only a fraction of all possible images so that the penalty for using words gets not too big.

Because permutations are such a fundamental instrument for our computations, the speed depends very much on the quality and techniques of permutation implementations. For PermLib both elementary permutations and permutation words have been implemented and their usage can be exchanged almost transparently. A very important issue for permutation words is when words should be multiplied out. This is hard to decide automatically so it can and should be triggered by the user if necessary. At the moment PermLib multiplies words out only if they are used as group generators.

The current permutation word implementation also is unable to automatically clear generators that are no longer used in any word from its storage. Although some kind of smart pointer would seem suitable for this purpose, the author has not succeeded yet with a comparably fast implementation based on a smart pointer without losing the ability of fast permutation inversion. Nevertheless, an automatic resource deallocation could also be implemented for the current version by manual reference counting.

4.1.2 Partition

Leon referenced both in [Leo91, Leo97] an additional paper that was yet to come and dealt with implementation techniques for partition backtracking. However, this paper has never appeared, but in [Leo97, p. 130] one can find a sketch of a data structure Leon proposed for partitions. A quick analysis of the GAP source code reveals that GAP uses very similar structures. The data structures for partitions are a crucial element for a partition backtracking implementation to become fast so we take a look at current public state-of-the-art. Closed source-implementations like [`Magma`] may still do it differently.

An obvious requirement for a partition data structure in the context of backtracking is the ability to perform fast intersections. A less obvious requirement that the author experienced during his experiments also is fast intersection reverting, i. e. recovering the original partition Π from $\Pi' := \Pi \wedge \Sigma$. This is because copying complete partitions in memory with their $O(n)$ size as it would be necessary when diving into recursion is too expensive to yield competitive performance. Leon's approach for recovery mitigates this effect.

Leon proposes the use of seven arrays to represent a partition but a lower number is already sufficient. In the following we use four arrays to represent a partition $\Pi \in \mathrm{OP}(\Omega)$, where $\mathrm{OP}(\Omega)$ denotes the set of all ordered partitions of Ω. We set $n := |\Omega|$ and $k := |\Pi|$.

- `partition[1..n]` is an ordering of the numbers 1 to n where elements of the same cell are stored contiguously and that is consistent with the following three arrays.
- `cellSize[1..k]` contains at position i the size $|\Pi_i|$ of the i-th cell of Π.
- `cellStart[1..k]` contains at position i the index of the the first element of cell Π_i in `partition`. This means the i-th cell Π_i consists of `partition[cellStart[i]]`, `partition[cellStart[i] + 1]`, ..., `partition[cellStart[i] + cellSize[i] - 1]`.
- `cellOf[1..n]` contains at position i the cell number c such that $i \in \Pi_c$.

The array `cellOf` is not necessary to define a partition and is used only to enhance performance of recovering. Before we go into the details of recovery we start with intersection.

The intersection operation as defined in Definition 3.10 on page 37 is a convenient way to describe mathematically a refinement process but is not usable in an implementation. Usually only a small part of the $|\Pi| \cdot |\Sigma|$ intersections in $\Pi \wedge \Sigma$ have a non-trivial result. For instance, we can always skip all single-element cells of Π. Hence it is better from an algorithmic point of view to define an operation $\mathrm{intersect} : \mathrm{OP}(\Omega) \to \mathrm{OP}(\Omega)$ as

$$\mathrm{intersect}(\Pi, i, \Gamma) := \begin{cases} (\Pi_1, \ldots, \Pi_{i-1}, \Pi_i \cap \Gamma, \Pi_i \setminus \Gamma, \Pi_{i+1}, \ldots, \Pi_m) & \text{if } \emptyset \subsetneq \Pi_i \cap \Gamma \subsetneq \Pi_i \\ \Pi & \text{otherwise} \end{cases} \tag{4.1}$$

the intersection of the i-th cell of Π with a set $\Gamma \subseteq \Omega$. We then can write $\Pi \wedge \Sigma$ as a series of consecutive intersections for some indices $(i_1, j_1), \ldots, (i_l, j_l)$:

$$\Pi \wedge \Sigma := \mathrm{intersect}(\ldots(\mathrm{intersect}(\mathrm{intersect}(\Pi, i_1, \Sigma_{j_1}), i_2, \Sigma_{j_2}), \ldots, i_l, \Sigma_{j_l}) \tag{4.2}$$

Note that the $i_1, i_2, \ldots, i_l$ need not to be pairwise different when we have to intersect one cell Σ_s with different cells of Π. Back to our data structure, this means that for $\mathrm{intersect}$ we have to split up a cell of Π. To facilitate this operation we maintain that every contiguous segment of `partition[]` that corresponds to a cell of Π is sorted. Additionally, we require that the intersecting set Γ is also a sorted list. Under these conditions we can compute both $\Pi_i \cap \Gamma$ and $\Pi_i \setminus \Gamma$ fast in $O(|\Gamma| + |\Pi_i|)$ time (cf. [Knu98, Sec. 5.3.2]).

In `partition[]` we split the area corresponding to cell i into two cells by introducing a new cell index $m := |\Pi| + 1$, adjusting `cellStart`, `cellSize` and `cellOf` of i and m and moving elements in the `partition` area as required by the cell split. In doing so, we maintain that both emerging cells are still sorted. After $\Pi' := \mathrm{intersect}(\Pi, i, \Gamma)$ we thus have cells $\Pi'_i = \Pi_i \cap \Gamma$, $\Pi'_m = \Pi_i \setminus \Gamma$ and $\Pi'_j = \Pi_j$ for all other cells j. If we establish that $\Pi = \mathrm{intersect}(\Pi, i, \Gamma)$, we do not have to change anything. Technically, the refinement relation $\mathrm{intersect}(\Pi, i, \Gamma) \leq \Pi$ does not necessarily hold because the cell indices i, m need not to be adjacent as required by Definition 3.8 on page 37. But this is only a labeling issue, which we can omit in an implementation for efficiency reasons without violating theoretical properties established before. Figure 4.1 depicts an example of relevant parts of a partition data structure before and after an $\mathrm{intersection}$.

Our second requirement for the partition data structure is recovering. If we have the result $\Pi' := \mathrm{intersect}(\Pi, i, \Gamma)$ of a intersection and $\Pi' \lneq \Pi$ we can recover Π in the

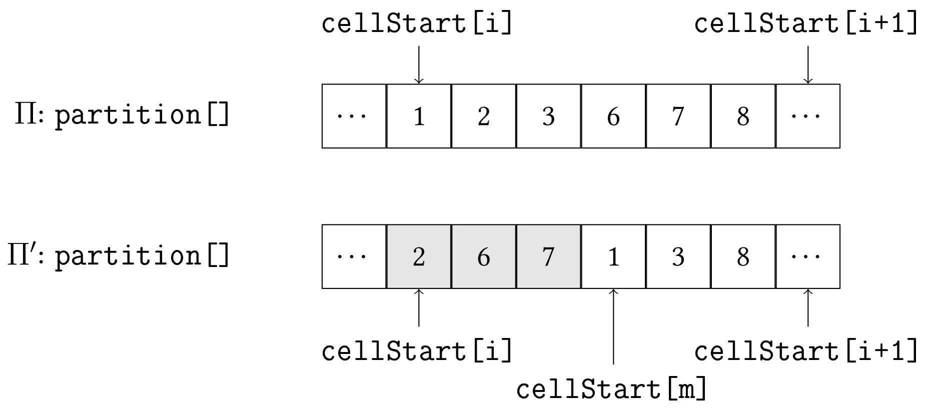


Figure 4.1: Example intersection of a partition $(\dots \mid 1\,2\,3\,6\,7\,8 \mid \dots)$ with $\{2, 6, 7\}$

following way. First we observe that $\mathrm{intersect}$ splits cell i into two neighborly cells i, m and $m = |\Pi'| = |\Pi| + 1$ as shown in Figure 4.1. So we locate cell m by looking at `cellStart[m]`. We obtain the index i_0 of the cell we have to merge m with by looking at the left neighbor of the cell m `cellOf[partition[cellStart[m]-1]]`, containing i_0. Then we merge `partition[cellStart[i_0]]` up to `partition[cellStart[m] + cellSize[m]]` into a single sorted cell and update `cellSize` and `cellOf` of i_0, yielding a data structure representing Π. Hence recovery can be accomplished in $O(|\Pi_{i_0}|)$ time.

4.2 Experiments

The PermLib implementation allows a simple exchange of many presented algorithms and data structures. The user may choose freely among

- two permutation representations (cf. Section 4.1.1),
- three transversal implementations (cf. Section 2.2.1 and 2.4),
- two transposition-based base change algorithms with two transposition algorithms and one base change by construction from scratch (cf. Section 2.5),
- two subgroup search algorithms with different pruning options (cf. Chapter 3),

which can all be combined transparently without interdependence.

This flexibility has the disadvantage that it leads to an explosion of the number of different configurations to test, over 120 in theory for a subgroup search. Because an extensive series of experiments covering all facets is beyond the scope and time limits of this thesis we proceed in two steps. In the first step we will gather evidence which combinations of permutation and transversals lead to promising results in base construction and base change experiments. We will also try to find out which base change algorithms lead to a good performance. In the second step we will use the top combinations of the first step to evaluate backtracking methods.

4.2.1 Data acquisition and setup

In the realm of groups it is hard to generate representative, random instances as groups come in many different flavors and structures. Therefore libraries of instances are very important if a broad and rather representative analysis is to be conducted. The groups we use for our analysis in this section come from three sources:

The open-source computer algebra system GAP contains various libraries. For our analysis we shall use the library of all primitive permutation groups because it offers the broadest range of order and degree (the smallest n for which a group is a subgroup of S_n) with many instances. From this library we will work with permutation groups up to degree 400 and with order $|G| \leq 2^{63}$, so that the group order easily fits into a 64-bit integer word. A group is called primitive if it preserves no nontrivial partition of G, i. e. for a set $\Delta \subseteq \Omega$ it holds for every $g \in G$ that $\Delta^g = \Delta$ and $\Delta^g \cap \Delta = \emptyset$ both imply $|\Delta| \in \{0, 1, |\Omega|\}$. The order limitation is not strictly necessary, it would suffice to bound the size of the fundamental orbits $\Delta^{(i)}$ in the base construction by 2^{63} for the code to work properly. The critical point is the orbit size comparison during the base transpose algorithms (Algorithm 2.7 and 2.8), which might get wrong if the size does no longer fit. But these sizes are a-priori unknown so we impose the limit on the group order and get a sample consisting of 2453 primitive groups over a quite broad degree range.

The second source is a small sample of 10 automorphism groups of polyhedra which the author has encountered over the last years. Naturally, this selection is not representative but may give indication of the usefulness of the discussed techniques for a designated application area.

The third source are full symmetric groups S_n for $n \in \{10, 20, 30, 40\}$. It is uncommon to search for set stabilizers in S_n or explicitly construct a BSGS for it because these are easily computable. But we include S_n into our experiments because the symmetric group plays an important in backtrack searches not covered in this thesis and may provide an extreme case of usage. For these we remove the 2^{63} order limit to work with halfway reasonable sized groups although there is a non-zero probability that the output may be wrong in some cases because integer overflows occur.

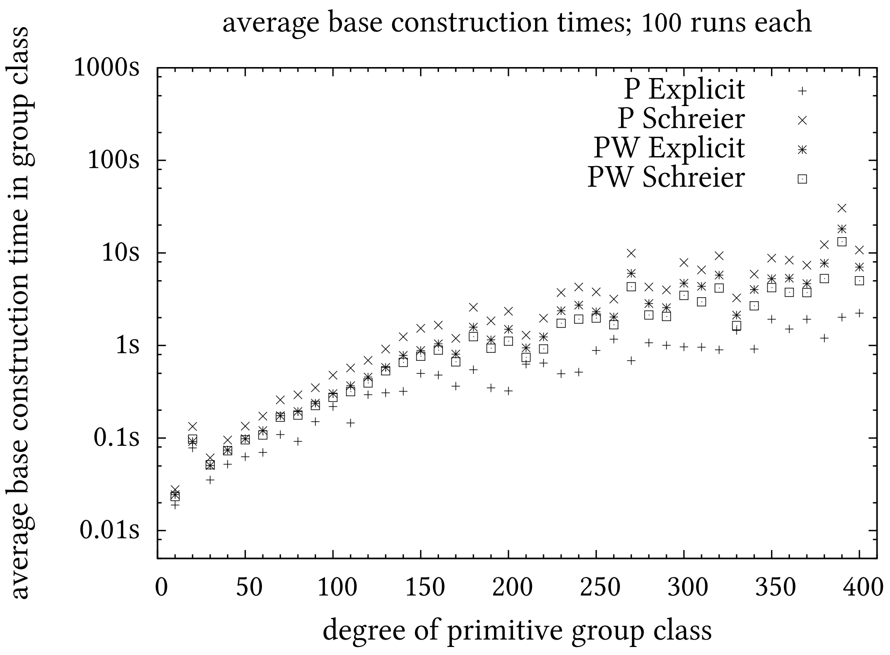
For the base change and set stabilization experiments we also need random subsets of Ω for prescribed bases or sets to stabilize. For these random subsets of Ω of given size were generated before so that all runs work on the same data.

All tests were conducted, if not stated otherwise, on the same machine, a server with four dual core AMD CPUs at 2 GHz each and 16 GB of RAM in total, running Ubuntu 8.04 in 64-bit version. The benchmark executables were compiled with the GNU C++ compiler in version 4.2.4 with `-O3` optimization setting and disabled asserts.

4.2.2 BSGS construction

For every group of the test set a base and a strong generating set have been computed multiple times with different parameter combinations. Parameters are the implementation of permutations (elementary or as words) and the transversal storage (explicit, as Schreier tree or as shallow Schreier tree).

Figure 4.2 shows the results for the library of primitive groups. For each group a BSGS was constructed 100 times. The results have been clustered with respect to the integer



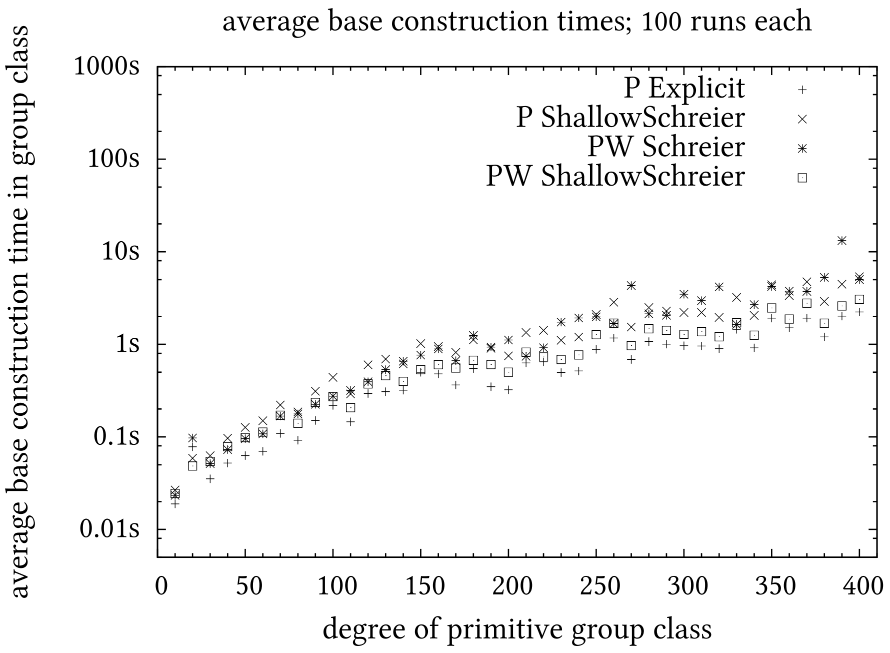


Figure 4.2: Base construction average times for primitive permutation groups

tenth of the degree, so degrees 10 to 19, 20 to 29, 30 to 39 &c. each form a class in the chart. Because the running time of the Schreier-Sims algorithm depends on both degree and order, this clustering regardless of the group structure is quite arbitrary and only used to find a presentable form. Throughout this chapter *P* and *PW* in a legend stand for elementary permutation implementation and implementation as permutation word, respectively. The second term in the legend specifies the used transversal implementation.

For almost all classes explicit transversals together with an elementary permutation implementation are the fastest combination. Elementary permutations used together with a Schreier tree transversal are throughout the sample the slowest combination. Permutation words work better with Schreier tree transversals but are still outperformed by elementary permutations and explicit transversals.

That explicit and Schreier tree transversals show very similar performance when used with permutation words is not by accident. Because permutations words in explicit transversals are usually not multiplied out but words in generators, "explicit" transversals for permutation words are just another form of Schreier tree storage and not really explicit. In this implementations the performance variation is caused by memory operation patterns which in this case favor explicit transversals slightly.

The reason why Schreier tree transversal implementations both with elementary permutations and permutation words are slowest has two different reasons. Because the Schreier-Sims construction mainly consists of building Schreier generators a lot of multiplications are needed to build the transversal elements from the tree. This high number of multiplications is the reason why the combination "P Schreier" is slow. We could already expect this from the asymptotic analysis of the Schreier-Sims algorithm on page 14. Permutations word, on the other hand, are less affected by the multiplications because for these word multiplications are relatively cheap. However, the words still have to be multiplied out to check if a Schreier generator is the identity. This occurs during the Schreier-Sims construction too often to actually gain a benefit.

It remains to analyze the performance of the shallow Schreier tree variant of Section 2.4 in the bottom part of Figure 4.2. For the primitive groups the shallow Schreier tree implementation using permutation words comes very close to the "P Explicit" optimum, and always at least as fast as the normal Schreier tree variant. The shallow trees together with elementary permutations gain over the normal Schreier trees (not in this figure), but are still inferior to the permutation words with the same transversal technique. These results are in accordance with our observations from above that the major factor of Schreier tree performance is the multiplications to construct a transversal element. Shallow Schreier trees reduce this number of edge label multiplications and are thus faster.

Figure 4.3 shows results for the Schreier-Sims algorithm implementation on the symmetric groups S_n for $n \in \{10, 20, 30, 40\}$. The algorithm configurations which the preliminary evidence of the primitive group library lets us suppose to be slow are omitted. Only the three most promising variants are shown and compared with GAP. The GAP calls used were as close to the API of the C++ implementation as possible, calling `StabChain(Group(...))` on a list of generators. The running times of PermLib are rather indifferent, with increasing degree the Shallow tree variant with permutation words has a minimal lead over the rest. At least for these examples the C++ implementation can compete with GAP performance. For this special case S_n, however, there are faster ways to construct a BSGS in GAP by calling `SymmetricGroup(n)` directly instead of `Group(.)`.

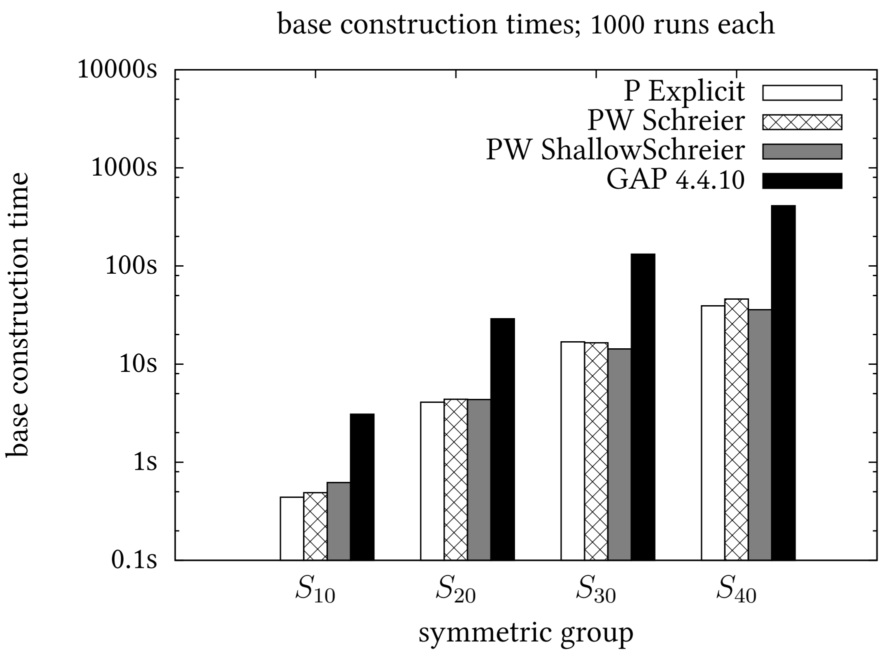


Figure 4.3: Base construction times for symmetric groups

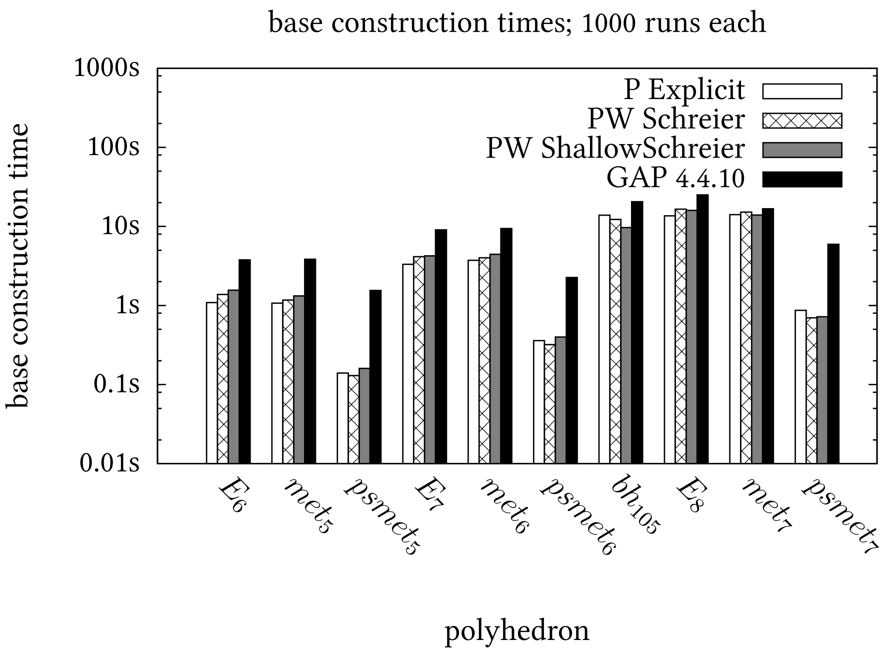


Figure 4.4: Base construction times for polyhedral automorphism groups

Using this specialization, GAP becomes even faster than the C++ implementation. However, this is no surprise because the set $\{(1\,2),(2\,3),\ldots,(n-1\,n)\}$ is a strong generating set relative to any $n-1$ points from Ω. Hence there is nothing really to compute and it only remains to set up the transversal data structures.

Finally, we look at the test set coming from practice, the polyhedral automorphism groups. Figure 4.4 shows the results of the runs with the top three performance parameter combinations and GAP. The automorphism groups are ordered by degree ascendingly from left (E_6 with degree 36) to right ($psmet_7$ with degree 154). From the PermLibvariations no one dominates the other. Rather a correlation between winning parameter combination and polyhedron class (E_x, met_x, $psmet_x$) can be suspected. The result pattern of GAP suggests there is a certain minimal time for initialization. At least this would explain why GAP comes closer to the C++ implementation for bigger instances than for the smaller ones.

4.2.3 Base change

Besides the two parameters from the last section, permutation and transversal implementation, we have two additional degrees of freedom for base change algorithms: the base change algorithm

- "simple", Algorithm 2.6 with transpositions only,
- "conjugation", Algorithm 2.9 with transposition and conjugation,
- "construction", Algorithm 2.10 re-construction from scratch,

and the base transposition algorithm for the transposition-based change algorithms.

To gather quickly evidence which combination works best with which permutation and transversal implementation we begin with experiments on a subset of the primitive group library. Instead of testing each of the 2453 groups from the library with a small number of base changes we perform many base changes on a smaller subset. We will use all primitive groups of degree 50, 100, 400 and a random selection of primitive groups of degree 256, which still gives us 109 groups to work with. Because we have to randomly generate prescribed bases for every degree that we want to test, this selection of four different degrees also helps to bound the experiment efforts. The numbers 50, 100, 256 and 400 themselves are not important, it is meant to be a not too small selection of groups with increasing degree.

First we examine the base transposition algorithms. Therefore we pick the base change algorithm 2.6 which uses only transpositions so that it is a good indicator for transposition performance. For the subset of primitive groups we plug in both the deterministic and the randomized transposition algorithm and repeatedly change bases according to a precomputed, randomly generated sequence of base points. Each of these 1000 prescribed bases has length $\log_2 n$ for a group of degree n, regardless of the actual group order.

Figure 4.5 shows the results of deterministic and randomized base point transposition side by side for four different permutation/transversal implementations. For both elementary permutation variations, the left four columns of each cluster in the figure, we see that the randomized version is measurably faster than the deterministic version. For the two permutation words combinations, the four columns on the right, there is only a small difference in the result between deterministic and randomized transposition. Analysis with

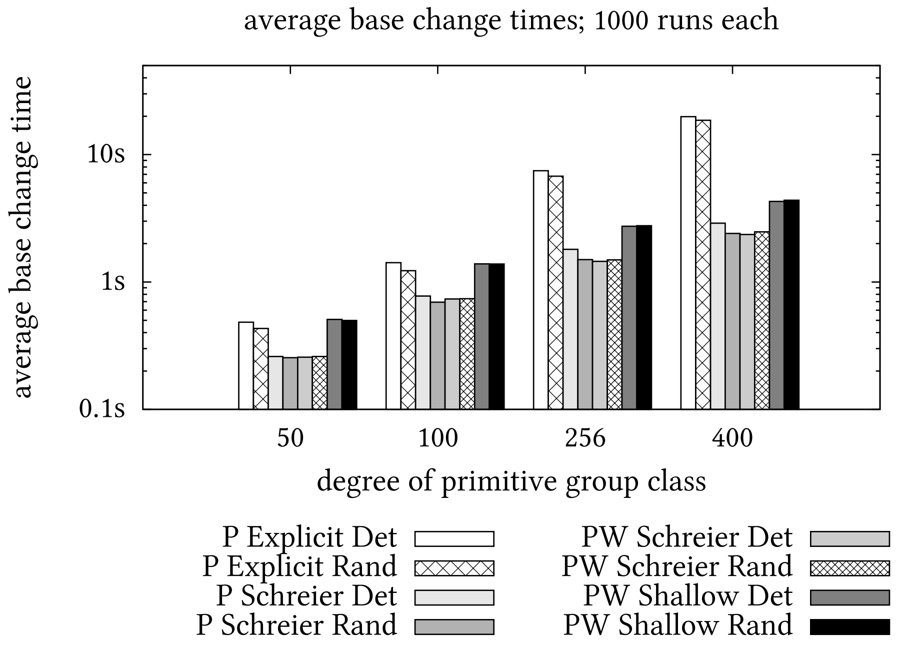


Figure 4.5: Base change/transposition average times for selected primitive groups

the performance analysis tool Valgrind [Vgrind] and internal statistics suggest that generating random Schreier generators based on permutation words is too slow to benefit from the reduced total generator number. For the remaining experiments we therefore use the deterministic base transpose algorithm for permutation words and the randomized version for elementary permutations.

The next step is to measure running times of the three base change algorithm. First we start with a similar setup: for each primitive group we change the base 1000 times according to a randomly generated list of bases of length $\lfloor \log_2 n \rfloor$. Figure 4.6 shows the results.

By comparing the left-most and right-most column of each cluster in the figure, we see that the "simple" base change with only transpositions performs always worse than its improved variant with conjugation. For most cases base change by construction, represented by the column in the middle, is the fastest method. This is not very surprising as we are in fact performing 1000 base constructions per group because the bases are totally unrelated. A more realistic test case would consist of sets that are more similar to each other and share the same prefix. This allows a better reuse of the previous BSGS elements.

Figure 4.7 shows the results of such a run with 1000 random bases of length $2\lfloor \log_2 n \rfloor$ that were each generated from their predecessor. Both transposition based algorithms run in almost zero time, whereas the construction method takes a lot of time. The absolute values of Figure 4.6 and 4.7 are not comparable because the prescribed base length differ and in the former run each base change was performed on a copy of a original BSGS and

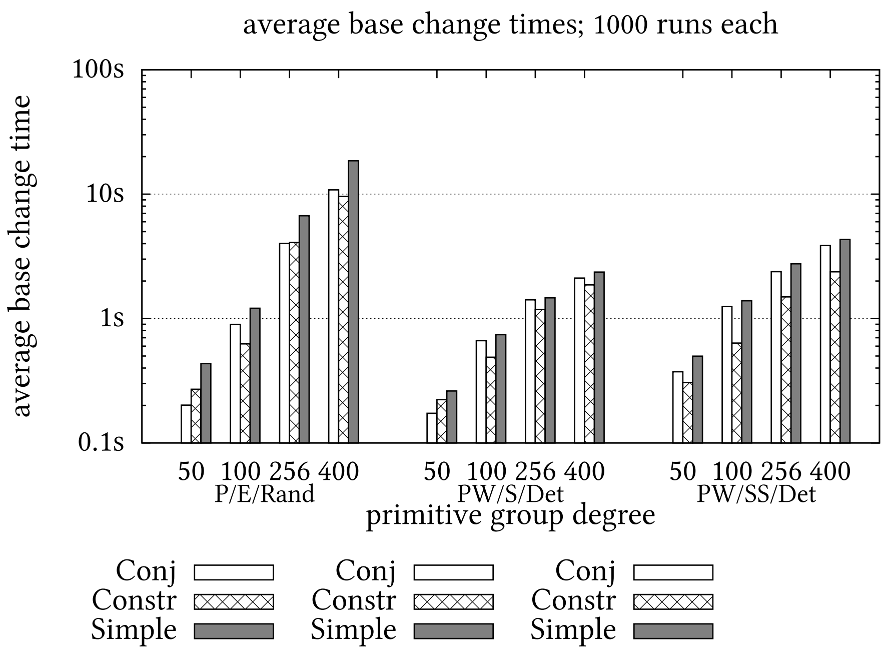


Figure 4.6: Base change average times for selected primitive groups

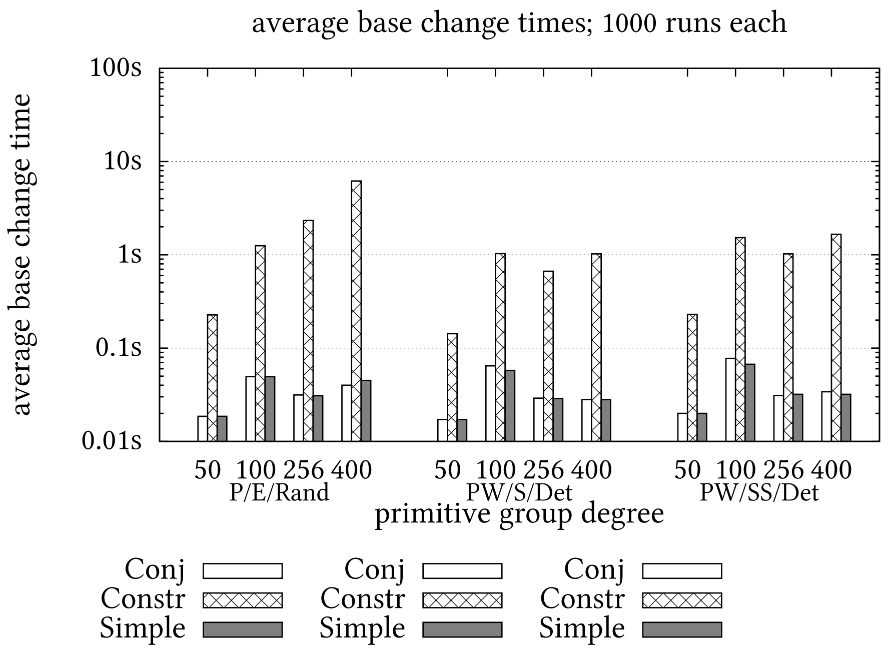


Figure 4.7: Base change average times for selected primitive groups and similar sets

in the latter run there was no such copy, so each base change ran on the result on its immediate predecessor to exploit the base similarity.

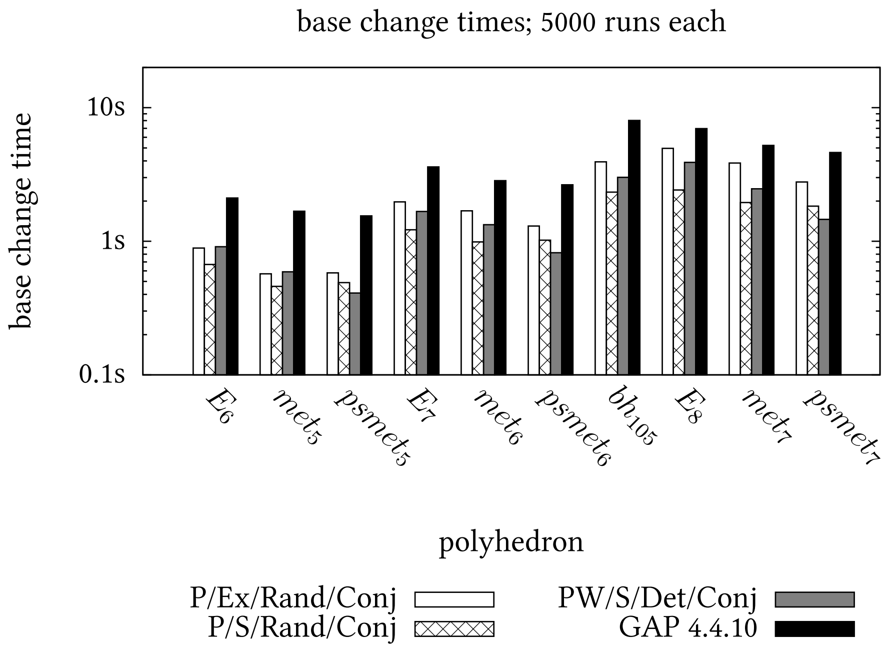


Figure 4.8: Base change times for polyhedral automorphism groups

Finally, we look at results from the polyhedron data set in Figure 4.8. For this experiment 5000 base changes have been executed with non-similar sets. We ignore the "construction" algorithm because its absolute performance in this test setup is not a good measure for practice as we have seen in Figure 4.7. Hence we compare the three best permutation/transversal combinations together with the "conjugation" algorithm with GAP. We look at the two middle columns of each cluster and observe that both Schreier tree transversals give the best performance. The reason for this is that conjugation is easier to perform on a Schreier tree transversal than on an explicit transversal. For a Schreier tree we only have to conjugate the group elements of the label set, i. e. the group generators. To update an explicit transversal after conjugation we have to conjugate every single transversal element. Usually the number of the latter is much larger than the number of group generators.

Again as in Figure 4.4 on page 56 we could guess a relationship between fastest permutation implementation (elementary or word) and the polyhedron class. The $psmet_x$ series seems to be the only one for which words are the faster choice. The results also show that the base change performance of PermLib can compete with GAP for these instances from practice.

4.2.4 Subgroup search: setwise stabilizer

When we analyze backtracking algorithms by setwise stabilizers we have one additional degree of freedom besides the choice of the backtrack algorithm: the form and size of the set to stabilize. Several thousand random subsets of Ω with different but prespecified length were generated for each group that is part of this analysis. Besides the backtracking algorithm, we can examine the effect of pruning. The correctness of the following results has been checked insofar as two different implementations always returned the same subgroup orders.

We begin with an experiment which gives some indication how the different parameters play together. Every group from the selection of primitive groups that we already have used for base change experiments in the last section stabilized 200 random generated sets of length $k-3, k-2, k-1, k, k+1$ each with $k = \lfloor \log_2 n \rfloor$. These length values were chosen heuristically so that not all set stabilizers become trivial. For presentation the average running times of all groups of same degree was computed and is shown in the following charts.

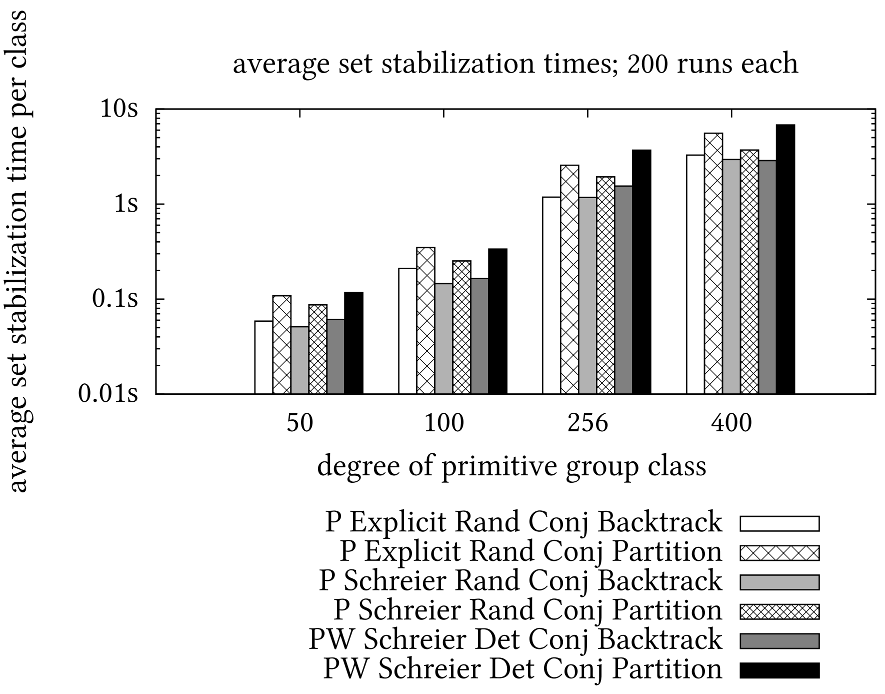


Figure 4.9: Set stabilizer search average times for primitive groups

Figure 4.9 depicts the running times of the two backtracking algorithm for three different permutation/transversal combinations. There is not a clear pattern which permutation/transversal combination performs best. But by comparing neighborly columns we observe that the classical backtracking algorithm is measurably faster than the partition backtracking variant for all three configurations. This is because the problem instances were solved by traversing only a quite small number of nodes. The pruning techniques of Section 3.1.4 prove to be efficient at least for small set sizes because the depth of the search tree is bounded by $|\Delta|$ and only nodes with $\gamma_i \in \Delta$ have to be considered. For larger sets we observe a different situation, though.

Figure 4.10 shows the results of a single group G of degree 100 and order about $1.2 \cdot 10^{12}$. For this run sets Δ of size 2 to 25 were randomly generated. For 50 sets of each size the stabilizer G_Δ was computed, for $|\Delta| \geq 13$ the group G_Δ had always order 2 or less. As shown for the small sets, classical backtracking beats partition backtracking because it has no overhead with partition handling for the small number of nodes. With increasing set size and also traversed node number the partition backtracking implementation becomes more and more efficient.

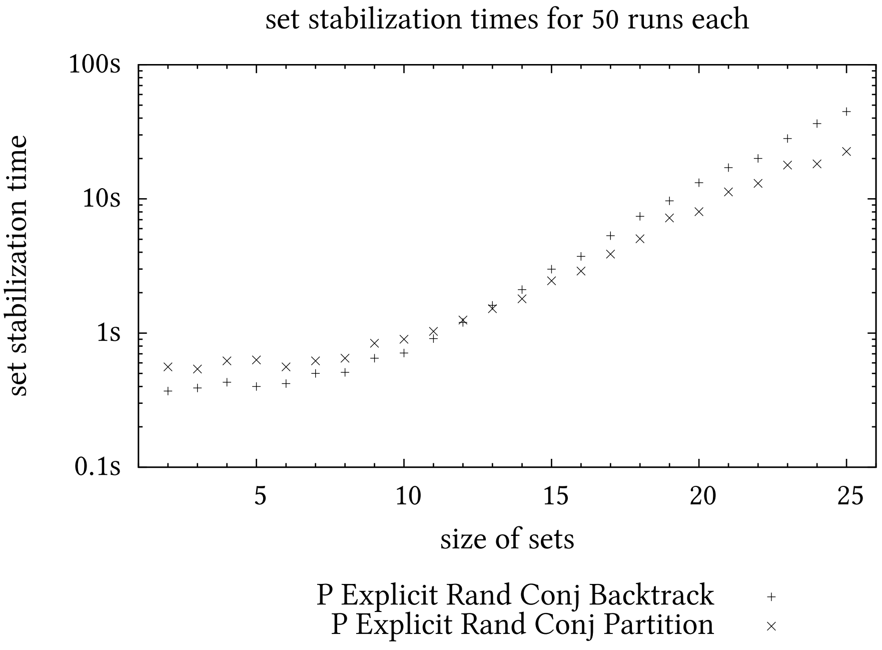


Figure 4.10: Set stabilizer search time depending on set size

In the next experiment we examine the influence of double coset minimality according to Lemma 3.4 on page 30. This is the only double coset minimality criterion we looked at that has significant costs because it possibly involves base changes. So all other presented criteria were enabled by default and we just toggle pruning based on Lemma 3.4, which we abbreviate with **DCM**. Figure 4.11 is a continuation of Figure 4.9 on page 61. We can see that DCM pruning has not a big, measurable effect on the running times. It tends to make the search slightly slower because of the additional base changes that are required. This, and the superior performance of classical backtracking for smaller set sizes, are an indication that the number of nodes is already quite near a minimum for backtrack search.

Finally, we compare PermLib with other implementations. We cannot compare with Leon's original code because it is not well-suited for sequential runs and sometimes caught segmentation faults during preliminary experiments. However, we can run experiments with the computer algebra software Magma [`Magma`], which states on its homepage that

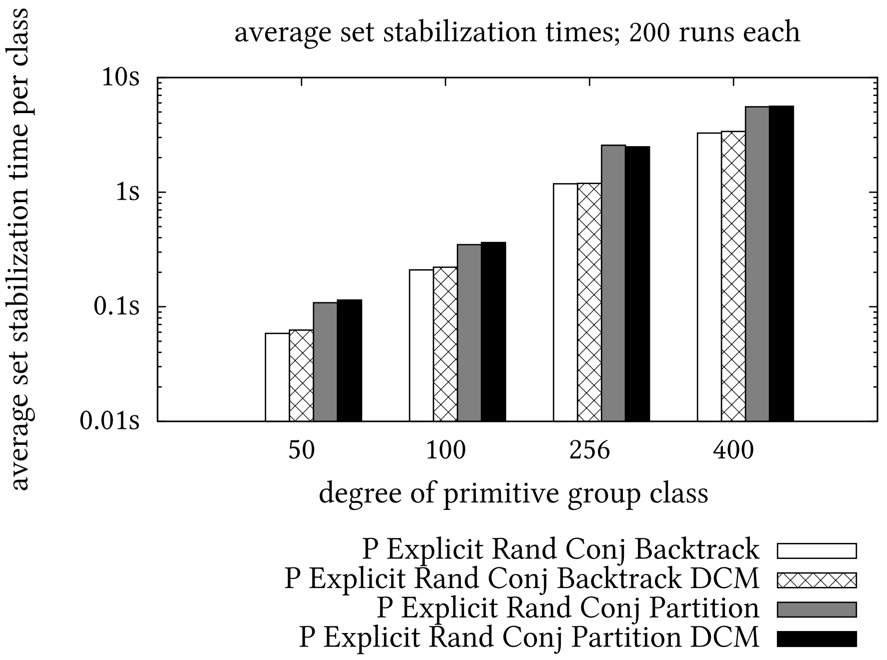


Figure 4.11: Set stabilizer search average times with enhanced double coset pruning for primitive groups

it uses code written by Leon for partition backtrack search. Because Magma is not freely available the following results were obtained on a different machine than the previous ones, so the absolute values are not comparable. Figure 4.12 and 4.13 show the running times for GAP and Magma and the fastest PermLib configuration based on elementary permutations and Schreier Tree transversals. For each polyhedron 5000 randomly generated sets of length $k-3, k-2, k-1, k, k+1$ each with $k = \lfloor \log_2 n \rfloor$ were stabilized, and 1000 for every symmetric group.

For both group types GAP, depicted by the right-most column in each cluster, is by far the slowest competitor in both cases. An interesting thing happens with Magma, related to the second-to-right columns. For the polyhedral groups it seems to scale a bit better than the author's implementation, shown in the left-most columns. However, in the runs with the symmetric groups its performance suddenly deteriorates significantly. Because it is unfortunately closed-source software we can only speculate about the cause. One reason could be that the stabilizer routine of Magma does not use a technique like we saw in Section 3.1.4 so that it does not find the generators $G_{(\Delta)} \leq G_\Delta$ early to prune the tree based on this subgroup or does not bound the search tree depth by $|\Delta|$. This seems to be a plausible assumption since neither [Leo91, Leo97] nor Leon's original C implementation (e. g. available through [`GUAVA`]) seem to contain this specialization.

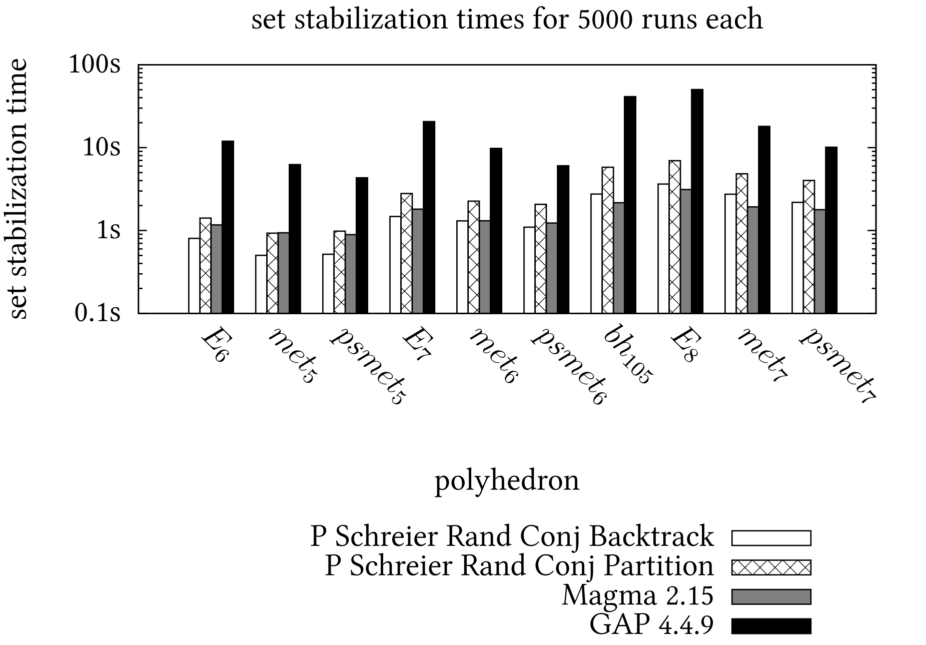


Figure 4.12: Set stabilizer search times for polyhedral automorphism groups

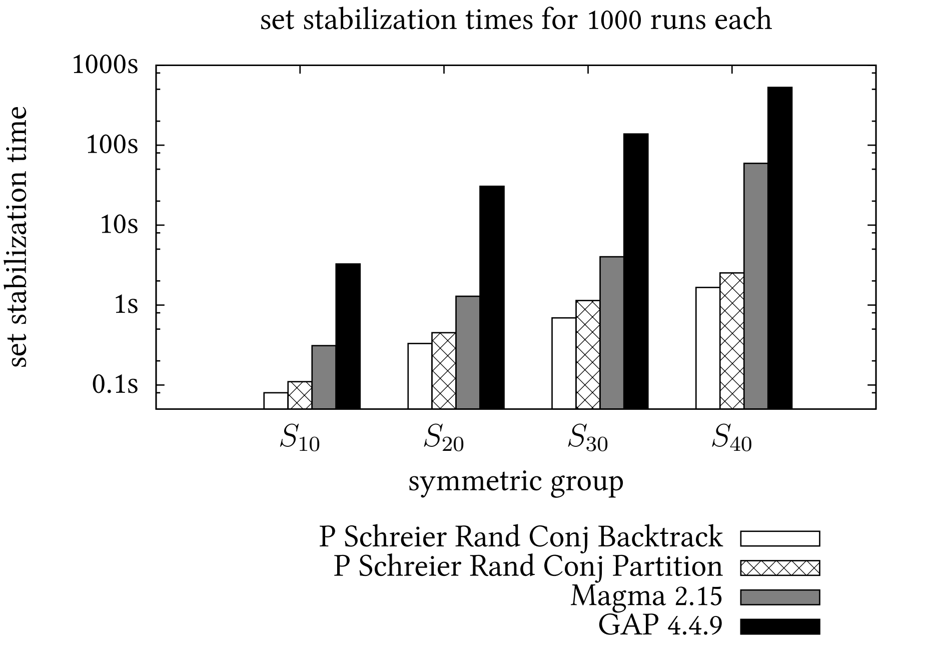


Figure 4.13: Set stabilizer search times for symmetric groups

4.2.5 Subgroup search: group intersection

Our final case study for backtracking subgroup searches are group intersections. The author has selected 20 primitive groups of different degree and order from the sample and built 10 intersecting pairs from it. All of these intersections except one (instance “9”) have an order greater than one.

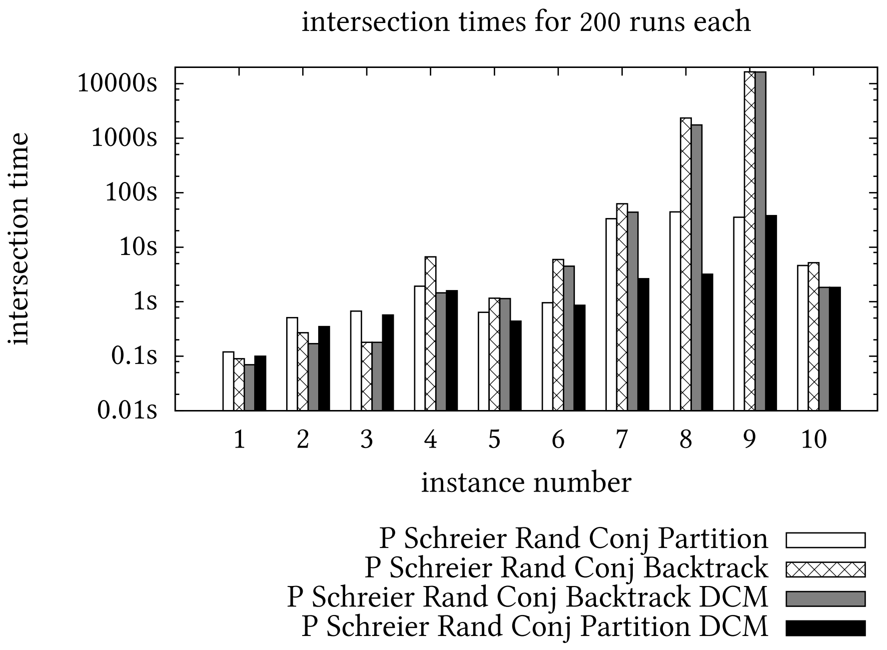


Figure 4.14: Intersection search times for primitive groups

Figure 4.14 shows the running times for intersections with both classical backtracking and partition backtracking, with and without DCM pruning. The long running times for some cases made tests with permutation words impossible because these are currently lacking proper memory management. So we restrict our analysis to elementary permutations with the fastest solution based on Schreier trees. The results are to some extent indifferent. Classical backtracking dominates the smaller instances, depicted by the two middle columns. Looking at the right-most columns, we see that partition backtracking with DCM pruning is fastest for some larger instances. Especially for the instance “9” classical backtracking takes 100 times longer to establish that the intersection is trivial.

Figure 4.15 helps us to understand this behavior. It shows the number of visited nodes during backtrack search in Figure 4.14. For most instances the absolute number differs not much between all parameter combinations, so classical backtracking may be faster because of smaller overhead. In the cases “7”, “8” and “9” partition backtracking together with DCM pruning can cut the number of nodes by the thousands.

In contrast to the set stabilizer experiments, DCM pruning can have a significant performance effect. This may be due to two reasons: First, the usefulness of DCM pruning depends on the size of the subgroup K. If K is trivial, we cannot prune by DCM.

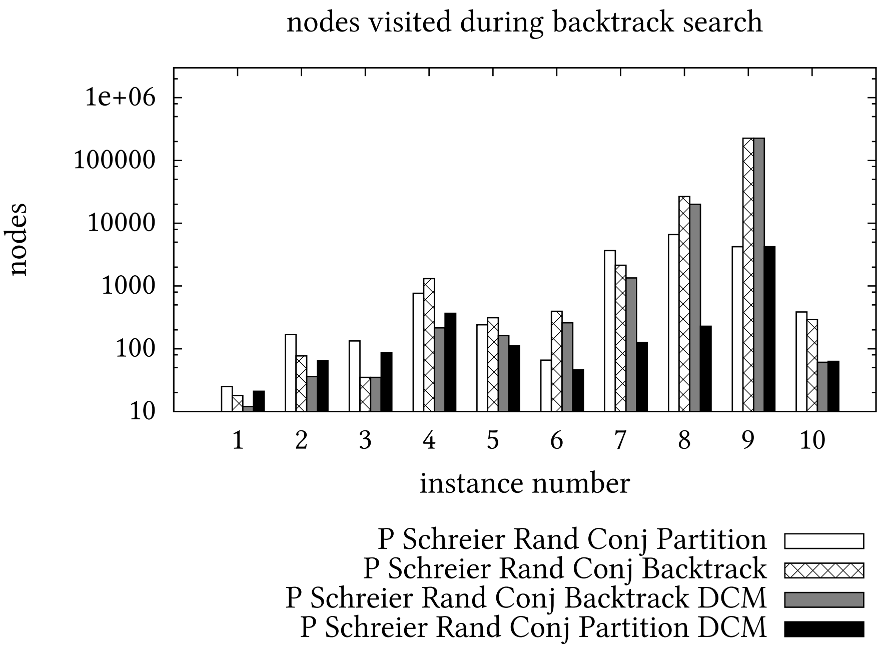


Figure 4.15: Number of nodes visited during intersection search for primitive groups

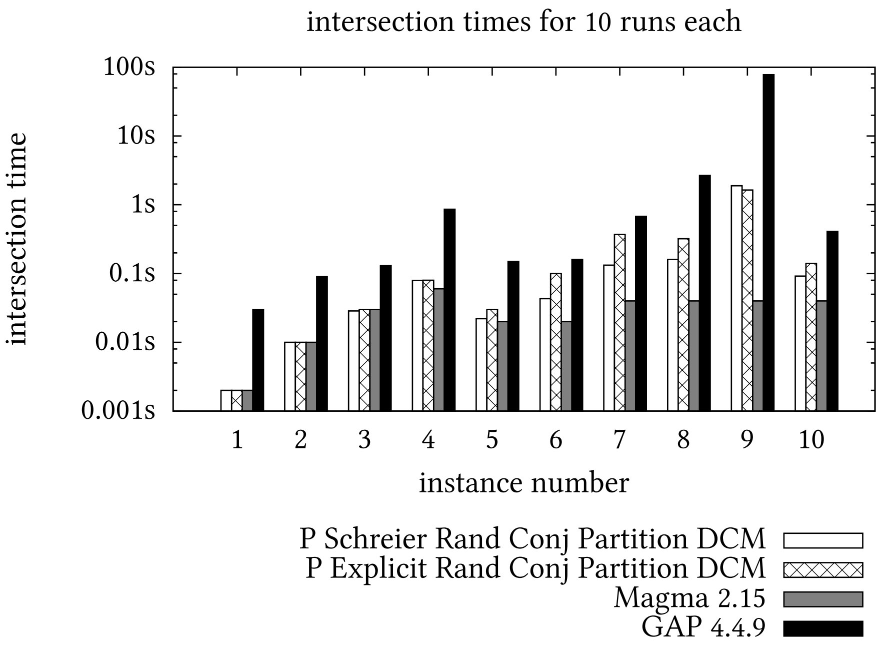


Figure 4.16: Intersection search times for primitive groups

So relatively big result groups compared to the set stabilizer instances can make DCM pruning more attractive. The second reason is restricted to partition backtracking: When computing intersections we usually start with a backtrack refinement on the empty partition (Ω) and have no small cell $(\Delta \mid \Omega \setminus \Delta)$ of size $|\Delta| \ll |\Omega|$ available to choose a first refinement from. This means that the first level of the search tree gets quite large and DCM pruning has a chance to mitigate this effect.

Finally, we compare again the performance with GAP and Magma in Figure 4.16. Similarly to the comparison of set stabilization performance, we can observe two facts: The performance of GAP is often worse than the author's implementation by a factor of ten. This may be due to the performance penalty caused by interpretation at runtime and traversing a large number of nodes. In contrast to that we see that Magma scales very well and runs almost in constant time throughout all instances. This means that it is in turn a factor of ten faster than the author's implementation for instance "9". The reasons of this huge differences are not clear.

4.2.6 Summary

We could observe that Schreier tree transversals are slower than explicit transversals for BSGS construction. Permutation words can mitigate the negative performance effect and lead to comparable running times in several cases. For base change there may be cases when a construction from scratch is faster than a change algorithm. From the "true" change algorithms the base change algorithm by conjugation, Algorithm 2.9 on page 22, works best. Randomized base point transpositions according to Algorithm 2.8 have a positive effect on the running time with elementary permutations. For permutation words there seems to be a slightly negative impact. Base changes on Schreier tree transversals are faster than changes on explicit transversals. Shallow Schreier trees seem to perform a base change slower than normal Schreier trees.

For the two subgroup search problems the combination of elementary permutations, Schreier tree transversal and base change by conjugation with randomized base transposition works best. There is no clear performance gain by using permutation words for the tested instances. Shallow Schreier trees also do not seem to have a positive impact on the backtrack running time. This may be due to worse base change performance compared to normal Schreier trees.

Small set stabilization instances with large stabilizer are solved fastest with a classical backtracking approach without extended pruning by double coset minimality. The larger the set to stabilize is, the more efficiently the partition backtracking approach works. It can be the fastest solution for large sets or small stabilizer. For intersection problems double coset minimality pruning is quite important. Small instances can be solved fast with classical backtracking. However, partition backtracking is not much slower and is even dramatically faster for several tested instances. Generally it seems reasonable to assume that classical backtracking can compete with partition backtracking if the ratio between $|G(\mathcal{P})|$ and $|G|$ is big enough because in this case $\mathcal{P}$-refinements cannot reduce the search space too much and the overhead of partition intersections becomes noticeable.

With a configuration of elementary permutation, Schreier tree transversal, base change by conjugation and randomized transpositions, PermLib is faster than GAP in every discipline except base construction. Because base construction happens usually only once this is no big disadvantage and can possibly be solved by further code optimization or using permutation words. There are huge running time differences in some set stabilization and intersection problems between GAP and PermLib. Magma is still a bit faster than PermLib on these examined backtracking problems. Especially for some intersection problems also larger differences appear. Nevertheless, there are also strange results with Magma and set stabilization in the symmetric group. It remains unclear what causes the observed dramatic differences in performance. Another interesting observation that the author has made in preliminary experiments is that the running times of PermLib do not scale equally between a 32-bit and a 64-bit platform. More experiments would be required to analyze this phenomenon.

5 Conclusion

5.1 Summary

In this thesis we have familiarized ourselves with some fundamental algorithms and data structures to tackle one basic problem in symmetry computation: search in permutation groups. The base and strong generating set data structure allows us to work efficiently with permutation groups on the computer. Besides the base and generators, transversals are a central part of a BSGS structure. We can use either an explicit or a Schreier tree variant to store transversal elements. With the Schreier-Sims algorithm we can construct a BSGS for a given permutation group. If in some context one base fits better than another we can choose between several base change algorithms. These dedicated base change algorithms are especially useful whenever the target base shares a common prefix with the source base.

A base and strong generating set structure induces a tree representation of a permutation group. We can use backtracking on this tree to search for elements with specific properties in this group. If the set of all sought elements is a subgroup or a coset we have powerful techniques at our disposal to exclude large portions of the tree from the search. For this pruning of the tree double coset minimality plays an important role. Furthermore, we have studied how we can specialize and bound the search space when we look for a setwise stabilizer or a group intersection.

We can structurally improve this backtrack search by using partitions and the concept of $\mathcal{P}$-refinements. These allow us to impose constraints at the same time on a set of elements instead of only a single point as classical backtracking does. We have examined $\mathcal{P}$-refinements for the setwise stabilizer and the group intersection/membership problem.

We also have looked at implementation details of permutations and partitions. Besides a trivial representation of permutations as vectors at the computer we can store permutations as words. Moreover, we have seen a data structure for partitions that is especially useful as it allows to go fast forwards and backwards during backtrack search.

The author has developed a C++ implementation called PermLib of all described algorithms and data structures. In the last section of this thesis we have analyzed the results of experiments with PermLib for BSGS construction, change and search algorithms. These results suggest that for the tested groups with degree up to 400 the representation of permutations as words does not significantly improve performance. The use of explicit transversals seems useful only for smaller groups and degree less than 50. We could observe the best overall performance with permutations stored as vectors and transversals stored in Schreier trees. The current PermLib implementation of partition-based backtracking is fast for group intersections and stabilizers of larger sets. For computing stabilizers of small sets classical backtracking seems to be the best approach.

The performance of PermLib is in all tested cases better than GAP with huge differences in some cases. Some problem instances also reveal a performance gap of PermLib towards the Magma implementation, which is often faster than PermLib.

5.2 Outlook

For future work it would be interesting to conduct in-depth experiments, which are able to perform a fair comparison of alternative algorithm implementations. For the comparison to GAP we have to keep in mind that it is interpreted at runtime and thus has a natural performance impediment. Some results in comparison to Magma are to some extent confusing and require more experiments to gather sufficient explanation of the observed phenomena.

From a practical point of view the author is especially interested in how his implementation performs in a real symmetry computation project and plans to use it for his upcoming study of polyhedral symmetry computation, which is currently bound to GAP and nauty. The author is convinced that the planned extension of PermLib towards computing polyhedral symmetries will show equally promising performance and that PermLib will serve as a useful library to support future work on exploiting symmetries.

A PermLib

PermLib currently has five main directories:

- `data` contains various permutation groups for testing or benchmarking.
- `doc` contains automatically generated documentation.
- `lib` contains the PermLib core. PermLib is completely implemented in C++ header files.
- `src` contains the applications which were used to benchmark PermLib.
- `test` contains various unit tests.

The only dependency to build PermLib is [Boost] in version 1.34.1 or higher. However, the build system for the contributed applications makes use of [CMake]. If CMake and Boost are correctly installed the PermLib applications can be built with:

```
~/permlib$ mkdir build && cd build
~/permlib/build$ cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo ..
~/permlib/build$ make
```

Then, for instance, `./src/exmaple` should run the example application below and compute a set stabilizer.

```
#include "permutation.h"
#include "bsgs.h"
#include "transversal/schreier_tree_transversal.h"
#include "construct/schreier_sims_construction.h"
#include "change/conjugating_base_change.h"
#include "search/classic/set_stabilizer_search.h"

#include <iostream>

int main(int argc, char *argv[]) {
  using namespace permlib;

  // we use elementary permutations
  typedef Permutation PERM;
  // and Schreier tree transversals
  typedef SchreierTreeTransversal<PERM> TRANSVERSAL;

  // our group will have degree 10
  const ulong n = 10;

  // group generators
  PERMlist groupGenerators;
  boost::shared_ptr<PERM> gen1(new PERM(n, std::string("1␣3␣5␣7␣9␣10␣2␣4␣6␣8")));
  groupGenerators.push_back(gen1);
  boost::shared_ptr<PERM> gen2(new PERM(n, std::string("1␣5")));
  groupGenerators.push_back(gen2);

  // BSGS construction
  SchreierSimsConstruction<PERM, TRANSVERSAL> schreierSims(n);
  BSGS<PERM, TRANSVERSAL> bsgs = schreierSims.construct(groupGenerators.begin(),
                                  groupGenerators.end());
  std::cout << "Group␣" << bsgs << std::endl;

  // we want to stabilize a set
  const ulong DeltaSize = 4;
  const ulong Delta[DeltaSize] = {0, 4, 7, 8};

  // change the base so that is prefixed by the set
  ConjugatingBaseChange<PERM,TRANSVERSAL,
    RandomBaseTranspose<PERM,TRANSVERSAL> > baseChange(bsgs);
  baseChange.change(bsgs, Delta, Delta+DeltaSize);

  // prepare search without DCM pruning
  classic::SetStabilizerSearch<PERM,TRANSVERSAL> backtrackSearch(bsgs, 0);
  backtrackSearch.construct(Delta, Delta+DeltaSize);

  // start the search
  BSGS<PERM,TRANSVERSAL> stabilizer(n);
  backtrackSearch.search(stabilizer);

  std::cout << "Stabilizer␣" << stabilizer << std::endl;

  return 0;
}
```

Nomenclature

$()$	identity permutation, page 3
α^G	orbit of $\alpha \in \Omega$ under $g \in G$, page 3
α^g	action of $g \in G$ on $\alpha \in \Omega$, page 3
$\mathrm{coset}\, n$	the coset which a node n of a search tree corresponds to, page 27
$\Delta^{(i)}$	i-th fundamental orbit, page 6
$\mathrm{fix}\, \Pi$	ordered sequence of single-element cells of a partition Π in the order in which they appeared in the refinement process, page 39
$\mathrm{intersect}$	intersection of a partition with a set, page 51
g^-	inverse of $g \in G$, page 3
$\log$	logarithm to base 2, page 6
$\lvert G : H \rvert$	index of H in G, page 3
$\lvert G \rvert$	order of group G, page 3
$\Omega_{\mathcal{P}}(n)$	child restriction in classical backtrack search of node n depending on property $\mathcal{P}$, page 31
$\mathrm{OP}(\Omega)$	set of all ordered partitions of Ω, page 37
$\Pi \wedge \Sigma$	intersection of two partitions Π, Σ, page 37
$\mathcal{U}(G)$	uniform distribution on the set or group G, page 19
$G^{[i]}$	pointwise stabilizer of the $i-1$ first base elements, page 5
$G_{(\alpha_1,\ldots,\alpha_k)}$	pointwise stabilizer of $(\alpha_1, \ldots, \alpha_k)$ as a tuple, page 4
$G_{\{\alpha_1,\ldots,\alpha_k\}}$	setwise stabilizer of $\{\alpha_1, \ldots, \alpha_k\}$ as a set, page 4
S^g	conjugate of $S \subseteq G$ under $g \in G$, page 21
S_n	the symmetric group of n elements, page 4

References

Bibliography

[Bab91] László Babai. Local expansion of vertex-transitive graphs and random generation in finite groups. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 164–174, New York, NY, USA, 1991. ACM.

[BL85] Gregory Butler and Clement W. H. Lam. A general backtrack algorithm for the isomorphism problem of combinatorial objects. *Journal of Symbolic Computation*, 1(4):363–381, 1985.

[Bri00] Gunnar Brinkmann. Isomorphism rejection in structure generation programs. In *Discrete Mathematical Chemistry. DIMACS Series in Discrete Mathematical and Theoretical Computer Science*, volume 51, pages 25–38. American Mathematical Society, 2000.

[BSS09] David Bremner, Mathieu Dutour Sikiric, and Achill Schürmann. Polyhedral representation conversion up to symmetries. In David Avis, David Bremner, and Antoine Deza, editors, *Polyhedral computation*, CRM Proceedings & Lecture Notes, pages 45–72. American Mathematical Society, 2009. Available from World Wide Web: `http://arxiv.org/abs/math/0702239`.

[But91] Gregory Butler. *Fundamental algorithms for permutation groups*. Springer, 1991.

[CH92] John J. Cannon and George Havas. Algorithms for groups. *Australian Computer Journal*, 24(2):51–60, 1992.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[CST89] Peter J. Cameron, Ron Solomon, and Alexandre Turull. Chains of subgroups in symmetric groups. *Journal of Algebra*, (127):340–352, 1989.

[HEO05] Derek F. Holt, Bettina Eick, and Eamonn A. O'Brien. *Handbook of Computational Group Theory*. Discrete Mathematics and Applications. Chapman & Hall/CRC, 2005.

[Jun03] Tommi Junttila. On the symmetry reduction method for petri nets and similar formalisms. Research Report A80, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, September 2003.

[Ker99] Adalbert Kerber. *Applied Finite Group Actions*. Algorithms and Combinatorics. Springer, 2nd edition, 1999.

[Knu91] Donald E. Knuth. Efficient representation of perm groups. *Combinatorica*, 11(1):33–43, 1991.

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: sorting and searching.* Addison Wesley, 2nd edition, 1998.

[KÖ06] Petteri Kaski and Patric R.J. Östergård. *Classification Algorithms for Codes and Designs*, volume 15 of *Algorithms and Computation in Mathematics*. Springer, 2006.

[Leo80] Jeffrey S. Leon. On an algorithm for finding a base and a strong generating set for a group given by generating permutations. *Mathematics of Computation*, 35(151):941–974, July 1980. Available from World Wide Web: `http://www.jstor.org/pss/2006206`.

[Leo84] Jeffrey S. Leon. Computing automorphism groups of combinatorial objects. In *Computational group theory (Durham, 1982)*, pages 321–335, London, 1984. Academic Press.

[Leo91] Jeffrey S. Leon. Permutation group algorithms based on partitions, I: Theory and algorithms. *Journal of Symbolic Computation*, 12:533–583, 1991.

[Leo97] Jeffrey S. Leon. Partitions, refinements, and permutation group computation. In Larry Finkelstein and William M. Kantor, editors, *Groups and computation II*, volume 28 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 123–158. American Mathematical Society, Providence, R.I., 1997.

[Luk93] Eugene M. Luks. Permutation groups and polynomial-time computing. In *Groups and Computation, DIMACS series in Discrete Mathematics and Theoretical Computer Science*, volume 11, pages 139–175. American Mathematical Society, 1993.

[Mar09] François Margot. *50 Years of Integer Programming 1958-2008*, chapter Symmetry in Integer Linear Programming, pages 647–686. Springer, 2009.

[McK81] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981. Available from World Wide Web: `http://cs.anu.edu.au/~bdm/papers/pgi.pdf`.

[McK98] Brendan D. McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26(2):306–324, 1998. Available from World Wide Web: `http://cs.anu.edu.au/~bdm/papers/orderly.pdf`.

[Mila] Wiki of Robert L.Miller. Available from World Wide Web: `http://wiki.rlmiller.org/PermutationGroups`. [Online; accessed January 10, 2010].

[Milb] Robert L. Miller. rlm-blog. Available from World Wide Web: `http://blog.rlmiller.org/`. [Online; accessed January 10, 2010].

[Ser03] Ákos Seress. *Permutation Group Algorithms.* Cambridge University Press, 2003.

[Sim70] Charles C. Sims. Computational methods in the study of permutation groups. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 169–183. Pergamon Press, 1970.

[Sim71a] Charles C. Sims. Computation with permutation groups. In *SYMSAC '71: Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 23–28, New York, NY, USA, 1971. ACM.

[Sim71b] Charles C. Sims. Determining the conjugacy classes of a permutation group. In Garrett Birkhoff and Marshall Hall Jr., editors, *Computers in Algebra and Number Theory*, volume 4 of *SIAM-AMS Proceedings*, pages 191–195, Providence, R.I., 1971. American Mathematical Society.

[The97] Heiko Theißen. *Eine Methode zur Normalisatorberechnung in Permutationsgruppen mit Anwendungen in der Konstruktion primitiver Gruppen*, volume 21 of *Aachener Beiträge zur Mathematik*. Verlag der Augustiner Buchhandlung, 1997. Ph.D. thesis at RWTH Aachen.

[Zie95] Günter M. Ziegler. *Lectures on polytopes*. Springer, New York, 1995.

Software

`[Boost]` Boost free peer-reviewed portable C++ source libraries. `http://www.boost.org/`.

`[CMake]` CMake – Cross Platform Make. `http://www.cmake.org/`.

`[GAP]` GAP – Groups, Algorithms, Programming – a System for Computational Discrete Algebra. `http://www.gap-system.org/`.

`[GUAVA]` GUAVA – a GAP package for computing with error-correcting codes. `http://www.gap-system.org/Packages/guava.html`.

`[Magma]` MAGMA Computational Algebra System. `http://magma.maths.usyd.edu.au/`.

`[nauty]` nauty, computing automorphism groups of graphs and digraphs. `http://cs.anu.edu.au/~bdm/nauty/`.

`[Sage]` Sage Mathematics Software. `http://www.sagemath.org/`.

`[Vgrind]` Valgrind – a GPL'd system for debugging and profiling Linux programs. `http://valgrind.org/`.

Index